Programmierhandbuch

VERSION 2.5

Borland® ObjectWindows®

Borland C++ 4.5

ObjectWindows® Programmierhandbuch

BORLAND

Copyright © 1987, 1994 by Borland International, Inc.
Alle Rechte vorbehalten.

Borland hat möglicherweise Patente oder Patentanmeldungen für Teile des hier behandelten Themas. Dieses Handbuch gibt Ihnen keine Rechte auf diese Patente.

Alle Borland-Produkte sind Warenzeichen oder eingetragene Warenzeichen von Borland International, Inc. Andere aufgeführte Waren- und Produktbezeichnungen sind Warenzeichen oder eingetragene Warenzeichen der jeweiligen Rechteinhaber.

Einschränkung der Gewährleistung:
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber:

Borland GmbH
Monzastraße 4c
63225 Langen

Satz, Layout & Acrobat-Bearbeitung:

ALFES Computers, Publishing & Graphics, Siegen

Printed in Ireland.

Inhalt

Einführung 23
ObjectWindows-Dokumentation 23
Aufbau des Programmierhandbuchs 24
Symbole und Typographie in diesem Handbuch 26

Kapitel 1
Übersicht zu ObjectWindows 27
Arbeiten mit Klassenhierarchien 27
Die Verwendung einer Klasse 27
Ableiten neuer Klassen 27
Kombinieren von Objektverhalten 28
Instantiierung von Klassen 28
Abstrakte Klassen 29
Vererbung von Elementen 29
Verschiedene Typen von Elementfunktionen 30
Virtuelle Funktionen 31
Nichtvirtuelle Funktionen 31
Abstrakte Funktionen 31
Standardplatzhalter-Funktionen 32
Objekttypologie 32
Fensterklassen 32
Fenster 33
Rahmenfenster 33
MDI-Fenster 33
Dekorierte Fenster 33
Dialogfensterklassen 33
Allgemeine Dialogfenster 33
Andere Dialoge 34
Klassen für Dialogelemente 34
Standarddialogelemente von Windows 34
Hilfselemente 34
Dekorationen 34
Klassen für Grafik 35
DC-Klassen 35
GDI-Objekte 35
Klassen zum Drucken 36
Klassen für Module und Anwendungen 36
Dok/View-Klassen 36
Verschiedene Klassen 37
Menüs 37
Zwischenablage 37

Kapitel 2
Eine Einführung in ObjectWindows . 39
Erste Schritte 39
Dateien für die Einführung 40
Schritt 1: Die grundlegende Anwendung . . 40
Weiterführende Informationen 42
Schritt 2: Behandlung von Windows-Ereignissen 42
Hinzufügen einer Fensterklasse 42
Hinzufügen einer Antworttabelle 43
Ereignisbehandlungsfunktionen 44
Gekapselte API-Aufrufe 46
Überschreiben der Funktion CanClose 46
Verwenden von TMyWindow als Hauptfenster 46
Weiterführende Informationen 47
Schritt 3: Schreiben in das Fenster 48
Konstruieren eines Gerätekontextes 48
Drucken in den Gerätekontext 48
Leeren des Fensters 49
Weiterführende Informationen 50
Schritt 4: Zeichnen in das Fenster 50
Hinzufügen neuer Ereignisse 50
Hinzufügen eines TClientDC-Zeigers 51
Initialisieren von DragDC 52

Aufräumen nach DragDC 52
Weiterführende Informationen 53
Schritt 5: Ändern der Liniendicke 53
Hinzufügen eines Stiftes 53
Initialisieren des Stiftes 53
Auswählen des Stiftes für DragDC . 54
Ändern der Stiftgröße 54
Konstruieren eines Eingabedialogfensters 55
Ausführen eines Eingabedialogfensters 55
Aufrufen von SetPenSize 56
Aufräumen von Pen 57
Weiterführende Informationen 57
Schritt 6: Zeichnen des Fensters und Hinzufügen von Menüs 58
Neuzeichnen des Fensters 58
Speichern der Zeichnung 58
TPoints 59
TPointsIterator 60
Verwenden der Array-Klassen 61
Die Funktion Paint 62
Menükommandos 64
Hinzufügen der Ereignisbezeichner . 64
Hinzufügen von Menü-Ressourcen . 65
Hinzufügen von Antworttabelleneinträgen 65
Hinzufügen von Ereignisbehandlungen 66
Implementieren der Ereignisbehandlungen 66
Weiterführende Informationen 66
Schritt 7: Verwenden allgemeiner Dialogfenster 67
Änderungen an TMyWindow 67
FileData 67
IsDirty 67
IsNewFile 68
Verbessern von CanClose 68
Die Funktion CmFileSave 69
Die Funktion CmFileOpen 70
Die Funktion CmFileSaveAs 70
Öffnen und Speichern von Zeichnungen 71
Die Funktion OpenFile 71
Die Funktion SaveFile 72
Die Funktion CmAbout 73
Weiterführende Informationen 74
Schritt 8: Hinzufügen mehrerer Linien 74
Die Klasse TLine 74
Das Array Tlines 75
Einfügen und Extrahieren von TLine-Objekten 76
Der Einfügeoperator << 76
Ausleseoperator >> 76
Erweitern von TMyWindow 77
Die Funktion Paint 78
Weiterführende Informationen 79
Schritt 9: Ändern von Stiften 79
Änderungen an der Klasse TLine 79
Stift-Zugriffsfunktionen 80
Die Funktion Draw 81
Einfüge- und Ausleseoperatoren ... 82
Änderungen an der Klasse TMyWindow 82
Die Funktion CmPenColor 82
Weiterführende Informationen 84
Schritt 10: Hinzufügen von Dekorationen . 84
Ändern des Hauptfensters 84
Erzeugen der Statuszeile 85
Erzeugen der Dialogelementleiste 86
Konstruieren von TControlBar 86
Erzeugen von Schalter-Gadgets 86
Separator-Gadgets 88
Einfügen von Gadgets in die Dialogelementleiste 88
Einfügen von Objekten in einen dekorierten Rahmen 89
Weiterführende Informationen 90
Schritt 11: Übergang zu MDI 90
Verstehen des MDI-Modells 90
Hinzufügen der MDI Header-Dateien 91
Ändern der Ressourcen-Script-Datei 91
Ersetzen der Rahmenfenster-Header-Datei 92
Hinzufügen von MDI-Client- und untergeordneten Header-Dateien 92
Ändern des Rahmenfensters 93

Erzeugen der MDI-Fensterklassen 95
Erzeugen der untergeordneten MDI-Fensterklasse 96
Deklaration der Klasse *TDrawMDIChild* 96
Erzeugen der Funktionen für *TDrawMDIChild* 97
Erzeugen des *TDrawMDIChild* Konstruktors 98
Erzeugen der MDI-Client-Fensterklasse 101
Funktionsumfang von *TMDIClient* 101
Datenelemente in *TDrawMDIClient* 102
Hinzufügen von Antwortfunktionen 102
GetFileData 104
Überschreiben von *InitChild* 104
Schritt 12: Übergang zum Dok/View-Modell 105
Organisieren des Quelltextes einer Anwendung 106
Das Dok/View-Modell 106
Die Klasse TDrawDocument 107
Erzeugen und Entfernen einer Instanz von TDrawDocument 107
Speichern von Zeilen 107
Implementieren virtueller TDocument-Funktionen 108
Öffnen und Schließen einer Darstellung 108
Speichern und Verwerfen von Änderungen 110
Zugriff auf die Daten eines Dokuments 113
Die Klasse TDrawView 114
TDrawView-Datenelemente 114
Erzeugen einer Instanz der TDrawView-Klasse 115
Benennen der Klasse 116
Protected-Funktionen 116
Ereignisbehandlung in TDrawView 117
Definieren von Dokument-Templates 118
Unterstützung von Dok/View in einer Anwendung 119
Die Funktion InitMainWindow 119
Die Funktion InitInstance 120
Funktionen zu TMyApp hinzufügen 120
Die Funtkion CmAbout 121
Die Funktion EvDropFiles 121
Die Funktion EvNewView 123
Die Funktion EvCloseView 124
Weiterführende Informationen 124
Schritt 13: Erstellen eines OLE-Containers 124
Wie OLE funktioniert 125
Was ist ein Container? 125
Implementierung von OLE in ObjectWindows: ObjectComponents 126
Hinzufügen von Header-Dateien für OLE-Klassen 127
Registrieren der Anwendung für OLE 128
Erzeugen eines Anwendungs-Dictionarys 131
Ändern von *TDrawApp* 132
Ändern der Klassendeklaration 132
Ändern der Klassenfunktionalität 133
Ändern der Dok/View-Klassen 135
Ändern der Dokumentregistrierung 136
Ändern von *TDrawDocument* zwecks Handhabung eingebetteter OLE-Objekte 137
Ändern von *TDrawView* zwecks Handhabung eingebetteter OLE-Objekte 142
Schritt 14: Erstellen eines OLE-Servers ... 147
Konvertieren Ihres Anwendungsobjekts ... 148
Ändern der Header-Dateien 148
Ändern der Registriertabelle der Anwendung 149

Ändern des Konstruktors der Anwendung 149
Verbergen des Hauptfensters des Servers 150
Identifizieren des Moduls 150
Erzeugen neuer Views 152
Ändern des übergeordneten Fensters des Info-Dialogfensters 154
Modifizieren von *OwlMain* 155
Ändern Ihrer Dok/View-Klassen 156
Ändern der Header-Dateien 156
Ändern der Dokumentregistriertabelle 156
Programmbezeichner und -beschreibung 157
Einfügen der Anwendung ermöglichen 157
Setzen der Menüeinträge des Servers 157
Spezifizieren von Zwischenablageformaten 158
Ändern der View-Meldungsfunktionen 159
Hinzufügen neuer Elemente zu *TDrawView* 161
Hinzufügen einer Mauspalette 161
Ausschneiden und Kopieren von Daten 161
Behandeln von ObjectComponents-Ereignissen 162
Entfernen von Aufrufen aus der Funktion *Paint* und den Mausfunktionen 167

Kapitel 3
Anwendungsobjekte 169

Die Minimalanforderungen 170
Miteinbinden der Header-Datei 170
Erzeugen eines Objektes 170
Finden des Objektes 171
Erzeugen der minimalen Anwendung 171
Initialisieren von Anwendungen 171
Konstruieren des Anwendungsobjektes 171
Verwenden von WinMain und OwlMain 172
Initialisieren der Anwendung 174
Initialisieren jeder neuen Instanz 175
Initialisieren der Hauptfenster 175
Festlegen des Anzeigemodus für das Hauptfenster 176
Wechseln des Hauptfensters 177
Botschaftsverarbeitung von Anwendungen 177
Gesonderte Botschaftsverarbeitung 177
Idle-Verarbeitung 177
Schließen von Anwendungen 178
Verhalten beim Schließen ändern 178
Die Anwendung schließen 179
Ändern von CanClose 179
Öffnen vordefinierter DLLs 179
Verwenden der Bibliothek für Dialogelemente im Borland-Stil (BWCC) 180
Verwenden der Bibliothek für 3D-Dialogelemente von Microsoft 180

Kapitel 4
Oberflächenobjekte 181

Warum Oberflächenobjekte? 182
Was bewirken Oberflächenobjekte? 182
Das generische Oberflächenobjekt: TWindow 182
Erzeugen von Oberflächenobjekten 183
Wann ist ein Fenster-Handle gültig? 183
Sichtbarmachen von Oberflächenelementen 184
Objekteigenschaften 185
Fenstereigenschaften 185
Entfernen von Oberflächenobjekten 186
Entfernen des Oberflächenelementes 186
Löschen des Oberflächenobjektes 187

Untergeordnete und übergeordnete Oberflächenelemente 187
Listen untergeordneter Fenster 188
Konstruieren von untergeordneten Fenstern 188
Erzeugen von untergeordneten Oberflächenelementen 189
Entfernen von Fenstern 190
Automatisches Erzeugen 190
Manipulieren von untergeordneten Fenstern 191
Operieren auf allen untergeordneten Fenstern: ForEach 191
Auffinden eines speziellen untergeordneten Fensters 192
Arbeiten mit der Liste untergeordneter Fenster 192
Registrieren von Fensterklassen 192

Kapitel 5
Ereignisbehandlung 193

Deklarieren von Antworttabellen 194
Definieren von Antworttabellen 194
Definieren der Antworttabelleneinträge ... 195
Kommandobotschaftsmakros 196
Windows-Botschaftsmakros 197
Benachrichtigungs-Botschaftsmakros mit der ID untergeordneter Fenster 199

Kapitel 6
Befehlsaktivierung 203

Befehlsaktivierungsbotschaften 203
Arbeiten mit Befehlsaktivierungsobjekten 206
ObjectWindows Befehlsaktivierungsobjekte 206
TCommandEnabler: die Befehlsaktivierungsschnittstelle 206
Funktionen 207
Datenelemente 208
Gewöhnliche Befehlsaktivierungstasks 209
Aktivierung und Deaktivierung von Befehlseinträgen 209
Text des Menüeintrags verändern 211
Umschalten von Befehlseinträgen 213
Schalter-Gadget Status 213

Kapitel 7
ObjectWindows Exception Handling 217

ObjectWindows Exception-Hierarchie 217
Arbeiten mit TXBase 218
Auf- und Abbau von TXBase 218
Klonen von Exception-Objekten 219
Auslösung von TXBase Exceptions 220
Arbeiten mit TXOwl 220
Auf- und Abbau von TXOwl 221
Klonen von TXOwl und von TXOwl abgeleiteten Exception-Objekten 221
Spezialisierte ObjectWindows Exception-Klassen 222
ObjectWindows Exception Handling-Makros 223
ObjectWindows Exceptions ein- und ausschalten 224
Makroexpansion 224

Kapitel 8
Fensterobjekte 227

Verwenden von Fensterobjekten 227
Konstruieren von Fensterobjekten 228
Einstellen von Attributen für die Erzeugung 229
Überschreiben von Standardattributen. 230
Attribute untergeordneter Fenster 230
Erzeugen von Oberflächenelementen für Fenster 231
Layoutfenster 232
Layoutbedingungen 233
Definieren von Bedingungen 233
Definieren bedingender Verhältnissen 236
Unvollständige Bedingungen 238

Verwenden von Layoutfenstern 238
Rahmenfenster 240
Konstruieren von Rahmenfensterobjekten 240
Konstruieren eines neuen Rahmenfensters 240
Konstruieren eines Rahmenfensteralias 241
Ändern von Rahmenfenstern 242
Dekorierte Rahmenfenster 243
Konstruieren von dekorierten Rahmenfensterobjekten 243
Hinzufügen von Dekorationen zu dekorierten Rahmenfenstern 244
MDI-Fenster 245
MDI-Anwendungen 245
Das MDI-Fenstermenü 245
Untergeordnete MDI-Fenster 245
MDI in ObjectWindows 245
Erzeugen einer MDI-Anwendung 246
Erzeugen eines MDI-Rahmenfensters 246
Erweitern des Verhaltens eines MDI-Client-Fensters 247
Manipulieren von untergeordneten Fenstern 247
Erzeugen eines untergeordneten MDI-Fensters 248
Automatisches Erzeugen untergeordneter Fenster 248
Manuelle Erzeugung untergeordneter Fenster 248

Kapitel 9
Menüobjekte 249

Konstruktion von Menüobjekten 249
Modifizieren von Menüobjekten 250
Abfrage von Menüobjekten 251
Verwendung von Menüobjekten 252
Verwendung von Pop-Up Menüobjekten .. 252
Verwendung von Menüobjekten mit Rahmenfenstern 254
Rahmenfenstern Menüquellen hinzufügen 254
Verwendung von Menüdeskriptoren 255
Erzeugen von Menüdarstellungen . 256
Konstruktion von Menüdeskriptor-Objekten 257
Erzeugen von Menügruppen in Menüquellen TMenuDescr Klasse;Gruppierung von Pop-Up Menüs 258
Zusammensetzung von Menüs mit Menüdeskriptoren 259

Kapitel 10
Dialogfensterobjekte 261

Verwenden von Dialogfensterobjekten 261
Konstruieren eines Dialogfensterobjektes 262
Aufruf des Konstruktors 262
Ausführen eines Dialogfensters 263
Modale Dialogfenster 263
Nicht-modale Dialogfenster 263
Automatisches Erzeugen von Dialogfenstern 265
Verwaltung von Dialogfenstern 266
Fehlerbehandlung beim Ausführen von Dialogfenstern 266
Schließen eines Dialogfensters 267
Verwenden von Dialogfenstern als Hauptfenster 268
Manipulieren von Dialogelementen in Dialogfenstern 269
Kommunikation mit Dialogelementen 269
Assoziieren von Oberflächenobjekten mit Dialogelementen 270
Dialogelementobjekte 270
Einrichten von Dialogelementen 271
Verwenden von Dialogfenstern 272
Verwenden von Eingabedialogfenstern 272
Verwenden von allgemeinen Dialogfenstern 273
Konstruieren eines allgemeinen Dialogfensters 273

Ausführen allgemeiner Dialogfenster 274
Verwenden allgemeiner Dialogfenster für Farbe 275
Verwenden allgemeiner Dialogfenster für Schrift 275
Verwenden allgemeiner Dialogfenster für Datei öffnen 277
Verwenden allgemeiner Dialogfenster für Datei sichern 278
Verwenden allgemeiner Dialogfenster für Suchen und Ersetzen 279
Konstruieren und Erzeugen von allgemeinen Dialogfenstern für Suchen und Ersetzen 279
Verarbeiten von Botschaften zum Suchen und Ersetzen 280
Behandeln des Kommandos »Weitersuchen« 281
Verwenden allgemeiner Dialogfenster für Drucker 281

Kapitel 11
Dok/View-Objekte 283

Wie Dokumente und Views zusammenarbeiten 283
Dokumentobjekte 285
View-Objekte 285
Dokument-Templates 286
Dokumentverwaltung 287
Dokument-Templates 288
Entwurf von Dokument-Template-Klassen 288
Erzeugen von Instanzen einer Template-Klasse 289
Ändern existierender Templates 290
Verwenden der Dokumentverwaltung 291
Konstruieren einer Dokumentverwaltung 292
Ereignisbehandlung von *TDocManager* 293
Erzeugen einer Dokumentklasse 295
Konstruieren von TDocument 295
Hinzufügen von Funktionalität zu Dokumenten 295
Funktionen für den Datenzugriff 296
Zugriff auf einen Stream 296
Stream-Liste 297
Komplexer Datenzugriff 297
Hilfsfunktionen für den Datenzugriff 298
Schließen eines Dokumentes 299
Erweitern der Dokumentfunktionalität 299
Arbeiten mit der Dokumentverwaltung 300
Arbeiten mit Views 300
Erzeugen einer View-Klasse 302
Konstruieren von TView 302
Hinzufügen von Funktionalität zu Views 303
Virtuelle Funktionen von TView 303
Hinzufügen eines Menüs 303
Hinzufügen einer Anzeige zu einem View 304
Hinzufügen von Zeigern auf Oberflächenobjekte 304
Vermischen von *TView* mit Oberflächenobjekten 304
Schließen eines View 305
Dok/View-Ereignisbehandlung 305
Dok/View-Ereignisbehandlung im Anwendungsobjekt 306
Dok/View-Ereignisbehandlung in einem View 307
Behandlung vordefinierter Dok/View-Ereignisse 307
Hinzufügen eigener View-Ereignisse 308
Dok/View-Eigenschaften 309
Eigenschaftswerte und -namen 310
Zugreifen auf Eigenschaftsinformationen 311
Ermitteln und Setzen von Eigenschaften 311

Kapitel 12
Dialogelementobjekte 313
Dialogelementklassen 313
Was sind Dialogelementobjekte? 314
Konstruieren und Entfernen von Dialogelementobjekten 315
Konstruieren von Dialogelementobjekten 315
Hinzufügen des Zeigers auf ein Dialogelementobjekt als Datenelement 315
Aufrufen des Konstruktors für Dialogelementobjekte 316
Ändern der Attribute von Dialogelementen 316
Initialisieren das Dialogelementes .. 317
Anzeigen von Dialogelementen 317
Entfernen des Dialogelementes 317
Kommunizieren mit Dialogelementobjekten 317
Manipulieren von Dialogelementen 318
Antworten auf Dialogelemente 318
Ändern des Verhaltens von Fenstern in Richtung auf Dialogfenster 318
Verwenden besonderer Dialogelemente ... 318
Verwenden der Listenfeld-Dialogelemente 318
Konstruieren eines Listenfeldobjektes 319
Ändern von Listenfeldern 319
Abfragen der Listenfelder 320
Antworten auf Listenfelder 320
Verwenden von statischen Dialogelementen 321
Konstruieren von statischen Dialogelementobjekten 321
Ändern statischer Dialogelemente .. 322
Abfragen statischer Dialogelemente 323
Verwenden von Schaltern als Dialogelemente 323
Konstruieren von Schaltern 323
Antworten auf Schalter 323
Verwenden von Markierungsfeldern und Schaltfeldern als Dialogelemente 324
Konstruieren von Markierungsfeldern und Schaltfeldern 324
Ändern von Auswahlfeldern 325
Abfragen von Auswahlfeldern 325
Verwenden von Gruppenfeldern 326
Konstruieren von Gruppenfeldern .. 326
Gruppieren von Dialogelementen .. 326
Antworten auf Gruppenfelder 326
Verwenden von Bildlaufleisten 326
Konstruieren von Bildlaufleisten ... 327
Kontrollieren des Bildlaufleistenbereiches 327
Kontrollieren des Blätterumfangs .. 327
Abfragen von Bildlaufleisten 328
Ändern von Bildlaufleisten 328
Antworten auf Botschaften von Bildlaufleisten 328
Vermeiden von Verfolgungsbotschaften für Positionszeiger 329
Spezialisieren des Bildlaufleistenverhaltens 329
Verwenden von Gleitreglern und Meßanzeigen 330
Verwenden von Editierfeldern 330
Konstruieren von Editierfeldern ... 330
Verwenden der Zwischenablage und des Bearbeiten-Menüs 331
Abfragen von Editierfeldern 332
Ändern von Editierfeldern 332
Verwenden von Kombinationsfenstern 333
Verschiedene Formen von Kombinationsfenstern 333
Auswählen eines Kombinationsfenstertyps 334
Konstruieren von Kombinationsfenstern 335
Ändern von Kombinationsfenstern . 335
Abfragen der Kombinationsfenster . 335

Einstellen und Lesen von Dialogelementwerten 336
Warum Transferpuffer verwenden? 336
Definieren des Transferpuffers 337
Transfer von Listenfeldern 338
Transfer von Kombinationsfenstern 338
Definieren des korrespondierenden Fensters oder Dialogfensters 339
Verwenden von Transfer bei einem Dialogfenster 339
Verwenden von Transfer bei einem Fenster 340
Transferieren der Daten 340
Transferieren von Daten zu einem Fenster 340
Transferieren von Daten von einem Dialogfenster 340
Transferieren von Daten von einem Fenster 340
Unterstützen von Transfer für spezielle Dialogelemente 341

Kapitel 13
Gadgets und Gadgetfensterobjekte . 343

Gadgets 343
Die Klasse TGadget 343
Erzeugen und Löschen einer Instanz der Klasse TGadget 344
Identifizieren eines Gadgets 344
Verändern und Zugreifen auf das Erscheinungsbild eines Gadget 345
Begrenzen eines Gadget 346
Größenanpassung für ein Gadget 346
Festlegen der Gadgetgröße 346
Anpassen der Farben des Gadget an die Systemfarben 347
Public-Datenelemente der Klasse TGadet 347
Aktivieren und Deaktivieren von Gadgets 347
Ableiten von TGadget 348
Initialisieren und Entfernen 348
Zeichnen eines Gadget 348
Ungültig Erklären und Aktualisieren eines Gadgets 349
Mausereignisse in einem Gadget 350
Gadgetklassen in ObjectWindows 351
Die Klasse TSeparatorGadget 351
Die Klasse TTextGadget 352
Erzeugen und Löschen einer Instanz der Klasse TTextGadget ... 352
Zugreifen auf den Gadgettext 352
Die Klasse TBitmapGadget 353
Erzeugen und Löschen einer Instanz der Klasse TBitmapGadget 353
Auswählen eines neuen Bildes 353
Einstellen der Systemfarben 354
Die Klasse TButtonGadget 354
Erzeugen und Löschen einer Instanz der Klasse TButtonGadget 354
Zugriff auf die Information eines Schaltergadget 355
Typ des Schaltergadget festlegen 356
Aktivieren eines Kommandos 356
Setzen der Systemfarben 356
Die Klasse TControlGadget 356
Erzeugen und Löschen einer Instanz der Klasse TControlGadget 357
Gadgetfenster 357
Erzeugen und Löschen einer Instanz der Klasse TGadgetWindow 357
Erstellen eines Gadgetfensters 358
Einfügen eines Gadgets in ein Gadgetfenster 358
Entfernen eines Gadget aus einem Gadgetfenster 359

Setzen von Seitenränder und Anordnungsrichtung in einem Fenster 359
Anordnen der Gadgets 359
Benachrichtigen des Fensters über die Größenänderung eines Gadget .. 360
Größenanpassung eines Gadgetfenster 360
Zugriff auf die Schriftart des Fensters 361
Erfassen der Maus in einem Gadget 361
Setzen des Hinweismodus 361
Behandlung von Idle-Aktionen 362
Durchsuchen von Gadgets 362
Ableiten der Klasse TGadgetWindow 363
Zeichnen eines Gadgetfenster 363
Größe und inneres Rechteck 363
Layouteinheiten 364
Antwortfunktionen von Botschaften 365
Gadgetfensterklassen in ObjectWindows .. 365
Die Klasse TControlBar 365
Die Klasse TMessageBar 366
Erzeugen und Löschen einer Instanz der Klasse TMessageBar 366
Angaben von Text für die Meldungszeile 366
Hinweistext angeben 366
Die Klasse TStatusBar 366
Erzeugen und Löschen einer Instanz der Klasse TStatusBar 367
Einfügen von Gadgets in eine Statuszeile 367
Anzeigen von Modusindikatoren .. 367
Anordnen von Gadgets in der Statuszeile 368
Die Klasse TToolBox 368
Erzeugen und Löschen einer Instanz der Klasse TToolBox 369
Wechseln von Toolbox-Dimensionen 369

Kapitel 14
Druckerobjekte 371
Erzeugen eines Druckerobjektes 372
Erzeugen eines Ausdrucksobjektes 373
Drucken von Fensterinhalten 375
Drucken eines Dokumentes 376
Setzen der Druckparameter 376
Zählen von Seiten 376
Drucken jeder Seite 376
Anzeigen weiterer Seiten 377
Weitere Betrachtungen zum Ausdrucken 377
Auswählen verschiedener Drucker 378
Auswählen eines Druckers 378
Konfigurieren des Druckers 378
Zuweisen eines speziellen Druckers 378

Kapitel 15
Grafikobjekte 379
GDI-Klassenorganisation 379
Änderungen an gekapselten GDI-Funktionen 380
Arbeiten mit Gerätekontexten 382
Die Klasse TDC 383
Konstruieren und Entfernen von *TDC* 384
DC-Operatoren 384
DC-Funktionen 385
Auswählen und Wiederherstellen von GDI-Objekten 386
Hilfsfunktionen zum Zeichnen 386
Farb- und Palettenfunktionen 387
Funktionen für Attribute zum Zeichnen 387
Viewport- und Fenster-Abbildungsfunktionen 387
Koordinatenfunktionen 388
Funktionen zum Clipping, für Bereiche und zum Aktualisieren von Rechtecken 388
MetaFile-Funktionen 388
Funktionen zur aktuellen Position .. 388
Schriftfunktionen 388

Pfadfunktionen 389
Ausgabefunktionen 389
Objektdatenelemente und Funktionen 390
Die Klasse TPen 391
Konstruieren von TPen 391
Zugriffe auf TPen 392
Die Klasse TBrush 393
Konstruieren von TBrush 393
Zugriffe auf *TBrush* 394
Die Klasse TFont 395
Konstruieren von TFont 395
Zugriffe auf TFont 396
Die Klasse TPalette 397
Konstruieren von TPalette 397
Zugriffe auf TPalette 398
Elementfunktionen 398
Erweitern von TPalette 399
Die Klasse *TBitmap* 400
Konstruieren von *TBitmap* 400
Zugriffe auf TBitmap 402
Elementfunktionen 402
Erweitern von *TBitmap* 403
Die Klasse TRegion 404
Konstruieren und Entfernen von TRegion 404
Zugriffe auf TRegion 406
Elementfunktionen 406
Operatoren 407
Die Klasse TIcon 410
Konstruieren von TIcon 410
Zugriffe auf TIcon 412
Die Klasse TCursor 412
Konstruieren von TCursor 412
Zugriffe auf TCursor 413
Die Klasse TDib 414
Konstruieren und Entfernen von TDib 414
Zugriffe auf *TDib* 416
Typkonvertierungen 416
Zugriffe auf interne Strukturen 416
Die Zwischenablage 416
DIB-Information 417
Arbeiten im PAL- oder RGB-Modus 417
Anpassen von Oberflächenfarben an Systemfarben 419
Erweitern von TDib 420

Kapitel 16
Objekte zur Eingabeprüfung 423

Die Standardprüfungsklassen 423
Prüfungsbasisklassen 424
Die Filterprüfungsklasse 424
Die Bereichprüfungsklasse 424
Die Nachschlageprüfungsklasse 425
Die Nachschlageprüfungsklasse für Strings 425
Die Bildprüfungsklasse 425
Verwenden von Datenprüfungen 426
Konstruieren eines Editierfeldobjektes 427
Konstruieren und Zuweisen von Prüfungsobjekten 427
Überschreiben von Elementfunktionen einer Prüfung 427
Die Elementfunktion *Valid* 428
Die Elementfunktion *IsValid* 428
Die Elementfunktion *IsValidInput* 428
Die Elementfunktion *Error* 429

Kapitel 17
Dialogelementobjekte von Visual Basic 431

Verwenden von VBX-Dialogelementen 431
Klassen der VBX-Dialogelemente 432
Die Klasse TVbxControl 433
Konstruktoren von TVbxControl 433
Implizites und explizites Konstruieren 434
Die Klasse TVbxEventHandler 435
Behandeln von VBX-Dialogelementbotschaften 436
Die Ereignisantworttabelle 436
Interpretieren eines Dialogelementereignisses 437

Ermitteln von Ereignisinformationen ... 438
Zugreifen auf ein VBX-Dialogelement ... 438
VBX-Dialogelementeigenschaften. ... 439
Ermitteln von Eigenschaftsinformationen ... 439
Abfragen der Dialogelementeigenschaften ... 440
Setzen von Dialogelementeigenschaften ... 441
Methoden von VBX-Dialogelementen ... 442

Kapitel 18
Dynamische Linkbibliotheken von ObjectWindows ... 443

DLL-Funktionen schreiben ... 443
Eintritts- und Austrittsfunktionen einer DLL ... 444
LibMain ... 444
WEP ... 445
DllEntryPoint ... 445
DLL-Funktionen exportieren ... 446
DLL-Funktionen importieren (aufrufen) ... 446
Gemeinsam genutzte ObjectWindows-Klassen schreiben ... 447
Definieren von gemeinsam genutzten Klassen ... 447
Das TModul-Objekt ... 448
ObjectWindows als DLL ... 449
Eine ObjectWindows-DLL aus einer Nicht-ObjectWindows-Anwendung aufrufen ... 449
Implizites und explizites Laden ... 450
Statische und dynamische Linkbibliotheken mischen ... 450

Kapitel 19
Unterstützung von OLE in Borland C++ ... 453

Was kann ObjectComponents? ... 453
Wo soll man anfangen? ... 454
Schreiben von Anwendungen ... 454
Erzeugen einer neuen Anwendung ... 454
Umwandeln einer Anwendung in einen OLE-Container ... 455
Umwandeln einer Anwendung in einen einbettenden und verknüpfenden Server ... 455
Hinzufügen von Automatisierungsunterstützung ... 455
Weitere nützliche Themen ... 456
Erlernen von ObjectComponents ... 456
Was ist OLE? ... 457
Verknüpfen und Einbetten ... 457
Automatisierung ... 457
Wie sieht OLE aus? ... 458
Einfügen eines Objekts ... 459
Direkte Bearbeitung eines Objekts im Containerdokument ... 460
Aktivieren, Deaktivieren und Auswählen eines Objekts ... 462
Auffinden von Objekt-Aktionsarten ... 463
Verknüpfen eines Objekts ... 464
Öffnen eines Objekts zum Bearbeiten ... 466
Was ist ObjectComponents ... 467
ObjectComponents unterstützte OLE 2 Merkmale ... 468
Verwenden von ObjectComponents ... 471
Überblick über Klassen und Botschaften ... 471
Verknüpfende und einbettende Klassen ... 471
Verknüpfungobjekte ... 472
Automatisierungsklassen ... 472
ObjectComponents Botschaften ... 473
Botschaften und Fenster ... 476
Neue ObjectWindows-OLE-Klassen ... 476
Exception Handling in Object-Components ... 477
TXOLe und OLE-Fehlercodes ... 478
Erzeugen einer ObjectComponents-Anwendung ... 479

Weitergeben von Dateien mit Ihrer Anwendung 480
Wie ObjectComponents arbeitet 481
Wie ObjectComponents mit OLE kommuniziert 481
Wie ObjectComponents mit Ihnen kommuniziert 482
Verbindungen verknüpfen und einbetten 482
Automatisierungsverbindungen 484
Programmierhilfen von ObjectComponents 485
Hilfsprogramme 486
Wo finde ich Informationen? 487
Bücher 487
Online-Hilfe 487
Beispielprogramme 488
Glossar der OLE-Begriffe 489

Kapitel 20
Erzeugen eines OLE-Containers 499

Eine Dok/View-Anwendung in einen OLE-Container umwandeln 499
1. Objekte mit OLE verknüpfen 501
Ableiten eines Anwendungsobjekts von TOcModule 501
Ableiten von OLE-Klassen 501
Erzeugen eines Anwendungsverzeichnisses 502
2. Registrierung eines Containers 503
Erzeugen von Registrierungstabellen 503
Einführung in Registrierungsmakros 506
Erzeugen eines Registrar-Objekts 507
3. Unterstützen von OLE-Kommandos 509
Einrichten des *Edit*-Menüs 509
Laden und Speichern von Verbunddokumenten 510
4. Erzeugen des Containers 512
Einbinden von OLE-Headern 512
Compilieren und Linken 512
Eine ObjectWindows-Anwendung in einen OLE-Container umwandeln 513
1. Erstellen der Anwendung 515
Definieren Sie ein Objekt für das Anwendungsverzeichnis. 515
Verändern der Anwendungsklasse 515
2. Registrierung eines Containers. 516
Erstellen von Registrierungstabellen. 516
Erstellen eines Registrar-Objekts 517
3. Erstellung des Client-Fensters 519
Von einer OLE-Klasse erben 519
Verzögerung der Erstellung des Client-Fensters in SDI-Anwendungen 520
Erstellen von ObjectComponents View-und Dokument-Objekten 521
Programmierung der Benutzeroberfläche 522
Die Behandlung von OLE-abhängigen Meldungen und Ereignissen 522
Unterstützung von ;Zusammenführung vonMenüzusammenführung 525
Aktualisieren des Menüs Bearbeiten 528
Eine Werkzeugleisten-ID zuweisen 529
5. Einen Container aufbauen 529
OLE Header einschließen 529
Kompilieren und Verbinden 529
Umwandeln eines C++ Programms in einen OLE-Container 529
1. Registrieren eines Containers 531
Aufbau einer Registrierungsungstabelle 531
Erzeugen des Registrar-Objekts 532
Erzeugung eines Hauptspeicherverwalters 533

2. Erzeugen eines Sichtfensters 534
Erzeugen, Größe ändern und Löschen des Sichtfensters 534
Erzeugen von TOcDocument und TOcView 535
Verarbeiten von WM_OCEVENT .. 537
Bearbeiten ausgewählter View-Ereignisse 538
Zeichnen des Dokuments 538
Aktivieren und Deaktivieren von Objekten 540
3. Programmieren des Hauptfensters 540
Erzeugen des Hauptfensters 540
Behandlung von WM_OCEVENT . 541
Behandlung von ausgesuchten Anwendungsereignissen 541
Behandlung von Standard-OLE-Menübefehlen 542
Erzeugen des Programms 543
Einbinden der ObjectComponents Header-Dateien 543
Compilieren und linken 543

Kapitel 21
Erzeugen eines OLE-Servers 545

Umwandeln einer Dok/View-Anwendung in einen OLE-Server 545
1. Verbinden von Objekten mit dem OLE-System 547
Erzeugen eines Anwendungsverzeichnisses 547
Ableiten des Anwendungsobjekts von TOcModule 547
Erben von OLE-Klassen 548
2. Registrieren eines OLE-Servers 549
Erzeugen von Registriertabellen ... 549
Erzeugen eines Registrar-Objekts .. 553
Verarbeiten der Kommandozeile ... 554
3. Darstellen, Laden und Speichern von Objekten 556
Mitteilung an den Client bei Änderung des Objekts 556
Laden und Speichern der Dokumente des Servers 557
4. Erzeugen des Servers 558
Einbinden der OLE-Header 558
Compilieren und Linken 558
Umwandeln einer ObjectWindows-Anwendung in einen OLE-Server 559
1. Registrieren des Servers 559
Erzeugen eines Anwendungs-Dictionarys 559
Erzeugen von Registriertabellen ... 560
Erzeugen der Dokumentliste 562
Erzeugen des Registrar-Objekts 563
2. Einrichten des Client-Fensters 564
Erzeugen von Helper-Objekten für ein Dokument 564
3. Modifizieren der Anwendungsklasse 566
Funktion der Dokumentenliste TRegLink 566
4. Erzeugen eines Servers 567
Einbinden der OLE-Header-Dateien 567
Umwandeln einer C++ Anwendung in einen OLE-Server 568
1. Erzeugen eines Speicherreservierungsobjekts 569
2. Registrieren der Anwendung 569
Erzeugen der Registriertabellen 569
Erzeugen der Dokumentliste 571
Erzeugen des Registrar-Objekts 571
Schreiben der Factory Callback-Funktion 573
3. Erzeugen eines View-Fensters 576
Erzeugen, Ändern der Größe und Löschen des View-Fensters 577
Erzeugen eines neuen Server-Dokuments 578
Behandlung von WM_OCEVENT . 579
Behandlung bestimmter View-Ereignisse 580
Grafische Anzeige des Dokuments . 581
4. Programmieren des Hauptfensters 581

Erzeugen des Hauptfensters 581
Behandlung von WM_OCEVENT 582
Behandlung einzelner Anwendungsereignisse 582
5. Erzeugen des Servers 583
Einbinden der Include-Dateien von ObjectComponents 583
Compilieren und Linken 583
Funktionsweise der Registrierung 583
Speichern von Informationen in der Registrierungsdatenbank 584
Registrieren lokalisierter Einträge 585
Registrieren von anwendungsspezifischen Einträgen 586
Erzeugen eines DLL-Servers 586
Vor- und Nachteile von DLL-Servern 587
Erzeugen eines DLL-Servers 588
Aktualisieren Ihrer Dokumentregistrierstruktur 588
Compilieren und Linken 590
Debuggen eines DLL-Servers 591
Tools für DLL-Server 592
Registrieren Ihres DLL-Servers 592
Starten Ihres DLL-Servers 593

Kapitel 22
Automatisierung von Anwendungen 595

Vorgehensweise zur Automatisierung einer Anwendung 596
Registrierung eines Automatisierungs-Servers 597
Erzeugung einer Registriertabelle 597
Erzeugung eines Registrarobjektes 599
Deklaration von automatisierbaren Methoden und Eigenschaften 600
Deklarationsmakros erstellen 601
Installation von wahlfreien Hooks für Prüf- und Filterungszwecke 603
Definition externer Methoden und Eigenschaften 604
Definitionsmakros 606
Datentypbezeichner in einer Automatisierungsdefinition 607
Zugänglichmachen von Daten zu Aufzählungszwecken 609
Erzeugung des Servers 611
Einbinden von Header-Dateien 611
Kompilierung und Verknüpfung ... 611
Funktionsoptimierung des Automatisierungs-Servers 611
Zusammenfassung mehrerer C++ Objekte zu einem einzigen OLE-Automatisierungsobjekt 612
Mitteilung an OLE über gelöschtes Objekt 613
Lokalisierung von Symbolnamen 614
Eintragung von Übersetzungen in das Ressourcenskript 615
Kennzeichnung von übersetzbaren Strings im Quelltext 616
Verwendung von XLAT-Ressourcen in ObjectComponents 618
Lokalisierung von Registrierstrings 619
Zugänglichmachen von Objektkollektionen 619
Erzeugen und Zugänglichmachen einer Kollektionsklasse 620
Implementierung eines Iterators für die Kollektion 623
Hinzufügung weiterer Elemente zur Kollektionsklasse 625
Erzeugung einer Typbibliothek 625

Anhang A
Konvertieren von ObjectWindows Code 627

Konvertieren Ihres Codes 628

Konvertieren nach Borland C++ 4.5 629
Konvertierungen mit OWLCVT 630
Kommandozeilensyntax von OWLCVT 631
Sicherheitskopien für Ihre alten Quelltextdateien 631
Verwenden von OWLCVT 631
Checkliste zur Konvertierung 633
Prozeduren zur Konvertierung 635
Behandlung von Botschaften und Ereignissen 635
Entfernen der DDVT-Funktionen ... 636
Namenskonventionen 637
Hinzufügen der Deklaration einer Ereignisantworttabelle 637
Hinzufügen der Definition einer Ereignisantworttabelle 637
Hinzufügen von Einträgen für die Ereignisantworttabelle 638
Antworten auf Kommandobotschaften 638
Antworten auf Botschaften mit untergeordneter Fenster-ID 639
Antworten auf Benachrichtigungsbotschaften 640
Antworten auf generelle Botschaften 640
Beispiel einer Ereignisantworttabelle 641
Ändern Ihrer Fensterobjekte 642
Konvertieren von Konstruktoren ... 643
Aufrufen von Windows-API-Funktionen 644
Ändern von Header-Dateien 645
Verwenden der neuen Positionen für Header-Dateien 645
Verwenden der neuen gestrafften Header-Dateien von ObjectWindows 646
ObjectWindows-Ressourcen 646
Compilieren von Ressourcen 647
Menü-Ressourcen 647
Konstruieren virtueller Basen 648
Umwandeln virtueller Basen zu abgeleiteten Typen 648
Überwechseln von Object-basierten Containern zur BIDS-Bibliothek 649
Streaming 650
Entfernte Inserter- und Extractor-Operatoren>>) 650
Implementieren von Streaming 650
MDI-Klassen 651
Bilden des Rahmens und des Client 652
Bilden eines untergeordneten Fensters 653
WB_MDICHILD 653
Umgelegte Funktionen 654
Ersetzen von ActiveChild durch GetActiveChild 654
Die Variable *MainWindow* 654
Verwenden eines Dialogfensters als Hauptfenster 655
Die Botschaftsverarbeitungsfunktionen von *TApplication* 655
Die Funktion GetModule 656
Die Funktionen DefXXXProc 657
Überschreiben 657
Verwenden von DefWndProc für registrierte Botschaften 658
Die Funktion Paint 658
Die Funktionen CloseWindow, ShutDownWindow und Destroy 658
Die Funktionen ForEach und FirstThat 659
Die Klassen TComboBoxData und TListBoxData 659
Die Klassen TEditWindow und TFileWindow 660
Verwenden des Beispiels OLDFILEW 661
Hinzufügen von TEditSearch- und TEditFile-Client-Fenstern 661
Die Klassen TSearchDialog und TFileDialog 662
Die Funktion ActivationResponse 662

Die Funktionen Before-DispatchHandler und After-DispatchHandler 662
Die Funktion DispatchAMessage 663
Generelle Botschaften 663
Umgang mit dem Parameter DefProc 663
Kommandobotschaften 664
Das Datenelement KBHandlerWnd 664
MAXPATH 664
Stilkonventionen 664
Ändern von WinMain in OwlMain 664
Datentypen und Namen 665
Ersetzen von MakeWindow durch Create 666
Ersetzen von ExecDialog durch Execute 667
Ermitteln der Anwendungs- und Modulinstanz 667
Definieren von WIN30, WIN31 und STRICT 667
Fehlerquellen 667
OWLCVT-Fehler 668
Compiler-Warnungen 668
Compiler-Fehler 668
Laufzeitfehler 669

Index 673

Abbildungen

Abbildung 1.1:
Die Vererbung bei *TDialog* 30

Abbildung 2.1:
SDI im Vergleich mit MDI bei der
Zeichenbrettanwendung 90

Abbildung 3.1:
Oberflächenobjekte und Oberflächenelemente 181

Abbildung 9.1: Menüdarstellungs-Anwendung ohne
geöffnetes untergeordnetes Fenster 256
Abbildung 9.2:
Menüdeskriptor-Anwendung mit geöffnetem
untergeordneten Fenster 256

Abbildung 11.1:
Dok/View-Modell-Diagramm 254

Abbildung 19.1:
Das Menü Bearbeiten im Beispielprogramm SdiOle . . . 459
Abbildung 19.2:
Das Dialogfenster Objekt einfügen 460
Abbildung 19.3:
Ein neu eingefügtes Objekt wird direkt bearbeitet 461
Abbildung 19.4:
Dasselbe Objekt nachdem es editiert wurde 461
Abbildung 19.5:
Die wiederhergesellte Benutzeroberfläche des Containers,
nachdem das Objekt deaktiviert wurde 463
Abbildung 19.6:
Das lokale Menü eines ausgewählten Objekts 464
Abbildung 19.7:
Das Dialogfenster Objekt einfügen vor dem Einfügen
eines verknüpften Objekts 465
Abbildung 19.8:
Die neue Liste der Aktionsarten für das gerade verknüpfte
Objekt 466
Abbildung 19.9:
Ein zum Bearbeiten geöffnetes Objekt 466
Abbildung 19.10: Wie Anwendungen mit OLE durch
ObjectComponents zusammenarbeiten 467
Abbildung 19.11: Wie Objekte Ihrer Anwendung mit
ObjectComponents zusammenarbeiten. 483
Abbildung 19.12: Wie TServedObject eine automatisierte
Klasse mit OLE verbindet. 485

Tabellen

Tabelle 1.1:
Die Vererbung von Datenelementen 30
Tabelle 1.2:
Gerätekontexte, die von ObjectWindows gekapselt werden 35
Tabelle 1.3:
GDI-Hilfsklassen 35

Tabelle 2.1:
Einträge und Bezeichner des MDI-Fenstermenüs 92
Tabelle 2.2:
Von *TMDIClient* bearbeitete Ereignisse 101

Tabelle 5.1:
Kommandobotschaftsmakros 196
Tabelle 5.2:
Botschaftsmakros 197
Tabelle 5.3:
Benachrichtigungsmakros mit der ID untergeordneter Fenster 199

Tabelle 7.1:
Spezialisierte Exception-Klassen 222
Tabelle 7.2:
ObjectWindows Exception Handling-Makroexpansion 224

Tabelle 8.1:
Attribute für die Erzeugung von Fenstern 230
Listing 8.1:
Überschreiben von Standardattributen in einem Fensterkonstruktor 230
Listing 8.2:
Überschreiben von Attributen eines untergeordneten Fensters im Konstruktor des übergeordneten Fensters 231
Tabelle 8.2:
Standardattribute für Fenster. 231
Tabelle 8.3:
Standardmenüverhalten für untergeordnete MDI-Fenster 247

Tabelle 9.1:
Menüobjekte erzeugende Konstruktoren von *TMenu* 250
Tabelle 9.2: Menüobjekte modifizierende Bausteine von *TMenu* 250
Tabelle 9.3:
Menüobjekte abfragende Bausteine von TMenu 251
Tabelle 9.4:
TMenuDescr Konstruktoren 257
Tabelle 10.1:
Die von ObjectWindows gekapselten Dialogfenster . . . 272
Tabelle 10.2:
TData-Elemente für allgemeine Dialogfenster 273
Tabelle 10.3:
Datenelemente von *TChooseColorDialog::TData* 275
Tabelle 10.4:
Datenelemente von *TChooseFontDialog::TData* 276
Tabelle 10.5:
Datenelemente von *TOpenSaveDialog ::TData* 277
Tabelle 10.6:
Datenelemente von *TPrintDialog::TData* 281

Tabelle 11.1:
Das *Datei*-Menü der Dokumentverwaltung 287
Tabelle 11.2:
Vordefinierte Dok/View-Ereignisbehandlungen 307
Tabelle 11.3:
Dok/View-Eigenschaftsattribute 310

Tabelle 12.1:
Dialogelemente und ihre ObjectWindows-Klassen 313
Tabelle 12.2:
Elementfunktionen von *TListBox* für das Ändern von Listenfeldern 319
Tabelle 12.3:
Elementfunktion von *TListBox* für die Abfrage von Listenfeldern 320
Tabelle 12.4:
Listenfeld-Benachrichtigungs-botschaften 321
Tabelle 12.5:
Elementfunktionen von *TCheckBox* für die Änderung von Auswahlfeldern 325
Tabelle 12.6:
Elementfunktionen von *TCheckBox* für die Abfrage von Auswahlfeldern 326
Tabelle 12.7:
Benachrichtigungscodes und Elementfunktionen von *TScrollBar* 329
Tabelle 12.8:
Elementfunktionen von *TEdit* und *Bearbeiten*-Menükommandos 331

Tabelle 12.9:
Elementfunktionen von *TEdit* für die Abfrage von Editierfeldern ... 332
Tabelle 12.10:
Elementfunktionen von *TEdit* für das Ändern von Editierfeldern ... 333
Tabelle 12.11:
Zusammenfassung von Kombinationsfenster eigenschaften ... 334
Tabelle 12.12:
Elementfunktionen von *TComboBox* für das Ändern eines Kombinationsfensters ... 335
Tabelle 12.13:
Elementfunktionen von *TComboBox* für das Abfragen eines Kombinationsfensters ... 336
Tabelle 12.14:
Transferpufferelemente für jeden Dialogelementtyp ... 337
Tabelle 12.15:
Datenelemente von *TListBoxData* ... 338
Tabelle 12.16:
Elementfunktionen von *TListBoxData* ... 338
Tabelle 12.17:
Datenelemente von *TComboBoxData* ... 338
Tabelle 12.18:
Elementfunktionen von *TComboBoxData* ... 339
Tabelle 12.19:
Transfer-Flag-Parameter ... 341

Tabelle 18.1:
Erlaubte Bibliothekskombinationen ... 451

Tabelle 19.1:
Wie man Containerunterstützung in eine existierende Anwendung einfügt ... 455
Tabelle 19.2:
Wie man Server in eine existierende Anwendung einfügt ... 455
Tabelle 19.3:
Einige verknüpfende und einbettende ObjectComponents Klassen ... 471
Tabelle 19.4:
Einige Automatisierungsklassen von ObjectComponents ... 472
Tabelle 19.5:
Anwendungsbotschaften für TOcApp-Clients ... 474
Tabelle 19.6:
Botschaften der Ausschnitte für TOcView und TOcRemView Clients ... 475
Tabelle 19.7:
Neue Klassen in ObjectWindows für die OLE-Unterstützung ... 476
Tabelle 19.8:
ObjectComponents Exception-Klassen ... 477
Tabelle 19.9:
Bilbiotheken für die Erstellung von ObjectComponents-Programmen ... 480
Tabelle 19.10:
Beschreibungen der ObjectComponents-Kapitel in diesem Buch ... 487
Tabelle 19.11:
Online-Hilfedateien mit Informationen über ObjectComponents und OLE. ... 488

Tabelle 20.1:
Nicht-OLE-Klassen und zugehörige Klassen, die OLE-Unterstützung hinzufügen ... 502
Tabelle 20.2:
Schlüssel, den ein Container registriert, um Linken und Einbetten zu unterstützen (Fortsetzung) ... 505
Tabelle 20.3:
Kommandos, die ein OLE-Container in sein Menü *Bearbeiten* eingibt ... 510
Tabelle 20.4:
Bibliotheken für die Erzeugung von ObjectComponents-Programmen ... 513
Tabelle 20.5:
Standard-Meldungs-Behandlungsroutine, die OLE Funktionen zur Verfügung stellen ... 522

Tabelle 21.1:
Schlüssel, die ein OLE-Server registriert ... 551
Tabelle 21.2:
Von ObjektComponents erkannte Kommandozeilenschalter ... 555

Tabelle 22.1:
Vom Automatisierungs-Server registrierte Schlüssel ... 598
Tabelle 22.2:
Automatisierungsdatentypen ... 608
Tabelle 22.3:
Aufzählbare C++ Typen und Automatisierungstypen, die sie zugänglich machen ... 610

Tabelle A.1:
Elementfunktionen für die Beantwortung von Botschaften und Einträge der Ereignisantworttabelle ... 638

Einführung

ObjectWindows 2.5 ist die Entwicklungsumgebung von Borland C++ bei Anwendungen für Windows 3.1, Win32 und Windows NT. ObjectWindows ermöglicht Ihnen eine schnelle und leichte Entwicklung vollwertiger Windows-Anwendungen. ObjectWindows 2.5 weist sich durch die folgenden Merkmale aus:

- Einfache Portabilität zwischen 16- und 32-Bit-Umgebungen
- Automatisiertes Extrahieren von Botschaften
- Robuste Behandlung von Exceptions und Fehlern
- Leichtes Portieren zu anderen Compilern und Umgebungen, da keine speziellen Compiler- oder Spracherweiterungen verwendet werden
- Kapselung von Windows-GDI-Objekten
- Dok/View-Klassen für eine einfache Datenabstraktion und -anzeige
- Drucker- und Druckvoranzeigeklassen
- Unterstützung für Dialogelemente von Visual Basic
- Eingabeprüfungen

ObjectWindows-Dokumentation

Die Dokumentation zu ObjectWindows 2.5 umfaßt das ObjectWindows *Programmierhandbuch* (dieses Handbuch), das ObjectWindows *Referenzhandbuch* sowie Abschnitte der *Kurzreferenz*.

Das ObjectWindows *Referenzhandbuch* enthält eine ausführliche alphabetische Liste und Beschreibung aller ObjectWindows-Klassen sowie deren Elementfunktionen, Datenelemente usw. Das ObjectWindows *Referenzhandbuch* dient damit als Referenz für spezielle technische Informationen über eine ObjectWindows-Klasse oder Funktion.

Die *Kurzreferenz* enthält Beschreibungen zu Kapselungen der ObjectWindows-Klassen zusammen mit einem Diagramm der ObjectWindows-Hierarchie. Die *Kurzreferenz* dient dem schnellen Nachschlagen der Verhältnisse der Klassen untereinander.

Aufbau des Programmierhandbuchs

Das ObjectWindows *Programmierhandbuch* ist in Themen aufgeteilt, die nach Arbeitsgebieten gegliedert sind. Sie beschreiben die Verwendung funktionaler Gruppen von ObjectWindows-Klassen zur Erledigung der verschiedenen Aufgaben. Das Handbuch ist wie folgt unterteilt:

Dieses Kapitel, **Einführung**, enthält eine kurze Beschreibung von ObjectWindows 2.5 und verweist Sie für weitere Informationen auf andere Kapitel dieses Buches.

Kapitel 1: Übersicht zu ObjectWindows zeigt eine kurze, informelle Übersicht über die ObjectWindows-Hierarchie.

Kapitel 2: Eine Einführung in ObjectWindows enthält eine Einführung in 15 Schritten, die verschiedene Eigenschaften von ObjectWindows 2.5 vorstellt.

Kapitel 3: Anwendungsobjekte beschreibt Anwendungsobjekte und die Anwendungsklasse *TApplication*.

Kapitel 4: Oberflächenobjekte erläutert die Verwendung von Oberflächenobjekten innerhalb des Programmiermodells von ObjectWindows 2.5. Oberflächenobjekte sind Instanzen von Klassen, die Fenster, Dialogfenster und Dialogelemente repräsentieren. Diese Klassen basieren auf der Klasse *TWindow*.

Kapitel 5: Ereignisbehandlung erklärt Antworttabellen, die die von ObjectWindows 2.5 unterstützte Methode zur Behandlung von Ereignissen darstellen.

Kapitel 6: Befehlsaktivierung beschreibt den Mechanismus von ObjectWindows 2.5 zur Befehlsaktivierung. Damit lassen sich Befehlselemente wie Menüauswahlen oder Schalter von Dialogelementleisten aktivieren und deaktivieren. Ebenso ist das Setzen von Menüeintragstexten oder das Markieren bzw. Demarkieren von Befehlselementen möglich.

Kapitel 7: ObjectWindows Exception Handling beschreibt den Exception Handling-Mechanismus von ObjectWindows 2.5.

Kapitel 8: Fensterobjekte beschreibt Fensterobjekte einschließlich der Verwendung von Rahmenfenstern, Layout-Fenstern, dekorierten Rahmenfenstern und MDI-Fenstern.

Kapitel 9: Menüobjekte geht näher auf die Arbeit mit Menüobjekten und die Klasse *TMenu* ein.

Kapitel 10: Dialogfensterobjekte erklärt die Verwendung von Dialogfensterobjekten (wie zum Beispiel *TDialog*-Objekte oder von *TDialog* abgeleitete Objekte) sowie diejenigen der allgemeinen Dialogfenster von Windows, die auf der Klasse *TCommonDialog* basieren.

Kapitel 11: Dok/View-Objekte beschreibt das Dok/View-Programmiermodell von ObjectWindows, das die Klassen *TDocument*, *TView* und *TDocManager* verwendet.

Kapitel 12: Dialogelementobjekte beschreibt die Verwendung der verschiedenen Dialogelemente wie Schalter, Listenfelder, Editierfelder usw.

Kapitel 13: Gadgets und Gadgetfensterobjekte beschreibt die Verwendung von Gadgets und Gadgetfensterobjekten.

Kapitel 14: Druckerobjekte erläutert die Verwendung der Klassen für den Drucker und für die Druckvoranzeige.

Kapitel 15: Grafikobjekte stellt die Klassen vor, die das Windows-GDI kapseln.

Kapitel 16: Objekte zur Eingabeprüfung beschreibt die Verwendung der Objekte zur Überprüfung der Eingabe in Editierfeldern.

Kapitel 17: Dialogelementobjekte von Visual Basic beschreibt die Verwendung der Dialogelemente von Visual Basic und der Klasse *TVbxControl* in Ihrer ObjectWindows-Anwendung.

Kapitel 18: Dynamische Linkbibliotheken von ObjectWindows erläutert die Verwendung der in ObjectWindows gekapselten dynamischen Linkbibliotheken (DLLs).

Kapitel 19: Unterstützung von OLE in Borland C++ präsentiert eine Übersicht über die ObjectComponents-Kapselung der OLE 2.5-Funktionalität.

Kapitel 20: Erzeugen eines OLE-Containers beschreibt, wie OLE-fähige Container-Dokumente erzeugt werden können, die OLE-Bestandteile enthalten können.

Kapitel 21: Erzeugen eines OLE-Servers beschreibt, welche Eigenschaften neben den Container-Eigenschaften einer OLE-Anwendung hinzugefügt werden müssen, damit man Server-Fähigkeiten erhält.

Kapitel 22: Automatisierung von Anwendungen beschreibt die für ein Programm notwendigen Aktionen, damit es durch andere Programme mittels Automatisierung kontrolliert werden kann.

Anhang A: Konvertieren von ObjectWindows 1.0-Code nach ObjectWindows 2.x geht näher auf die Konvertierung Ihrer ObjectWindows 1.0-Anwendung nach ObjectWindows 2.x ein.

Symbole und Typographie in diesem Handbuch

Fettschrift — Fettschrift kennzeichnet Schlüsselwörter der Programmiersprache (z.B. **char**, **switch** und **begin**) sowie Kommandozeilenoptionen (z.B. **-rn**).

Kursiv — Kursive Schrift wird bei Programmvariablen und Konstanten eingesetzt, die im Text erscheinen. Diese Typographie wird ebenso für die Betonung bestimmter Wörter verwendet, wie beispielsweise von neuen Begriffen.

`Schreibmaschine` — Schreibmaschine steht für Text, der in gleicher Form auf dem Bildschirm oder in einem Programm erscheint. Diese Schriftart wird auch für jede manuelle Eingabe (wie zum Beispiel `TD32` für das Starten des 32-Bit Turbo-Debugger) verwendet.

[*Taste***]** — Diese Typographie kennzeichnet eine Taste Ihrer Tastatur, wie beispielsweise in »Drücken Sie [*Esc*], um das Menü zu verlassen«.

[*Taste1***] + [***Taste2***]** — Tastenkombinationen, bei denen eine oder mehrere Tasten gleichzeitig gedrückt werden, werden durch [*Taste1*] + [*Taste2*] repräsentiert. Sie können zum Beispiel das Programm-Reset-Kommando ausführen, indem Sie die Taste [*Strg*] gedrückt halten und die Taste [*F2*] drücken (repräsentiert durch [*Strg*] + [*F2*]).

Menü/Kommando — Diese Kommandosequenz repräsentiert die Auswahl einer Menüoption. So repräsentiert zum Beispiel »*Datei/Öffnen*« das Kommando *Öffnen* im Dateimenü.

Dieses Symbol weist Sie auf besondere Informationen hin, die Sie beachten sollten.

Dieses Handbuch verwendet ferner die folgenden Symbole, um Abschnitte anzuzeigen, die eine spezielle Betriebssystemumgebung betreffen:

Windows 3.x

Windows 32s *und* NT

Windows 32s

Windows NT

K A P I T E L 1

Übersicht zu ObjectWindows

Dieses Kapitel enthält eine Übersicht über die Hierarchie von ObjectWindows 2.5. Es beschreibt die grundlegenden Gruppierungen von Klassen, die in ObjectWindows 2.5 vorhanden sind. Es beschreibt ferner, wie jeder Bestandteil mit den anderen zusammenpaßt, und verweist auf spezielle Kapitel, die die Verwendung einer spezifischen Klasse näher erläutern.

Arbeiten mit Klassenhierarchien

Dieser Abschnitt geht auf einige grundlegende Eigenschaften von Klassen, insbesondere auf diejenigen von ObjectWindows-Klassen, näher ein. Er enthält folgende Punkte:

- Die Verwendung einer Klasse
- Die Vererbung von Elementen
- Die Typen von Elementfunktionen

Die Verwendung einer Klasse

Es gibt drei grundlegende Arten, eine Klasse zu verwenden:

Man leitet von ihr eine neue Klasse ab.

- Man fügt ihr Verhalten zu dem einer anderen Klasse hinzu.
- Man erzeugt eine Instanz der Klasse (instantiiert sie).

Ableiten neuer Klassen

Um das Verhalten einer Klasse zu ändern oder neues Verhalten hinzuzufügen, müssen Sie von ihr eine neue Klasse ableiten:

```
class TNewWindow : public TFrameWindow {
public:
     TNewWindow(...);
     };
```

Mit dem Ableiten einer neuen Klasse können Sie drei mögliche Veränderungen vornehmen:

- Hinzufügen neuer Datenelemente
- Hinzufügen neuer Elementfunktionen
- Überschreiben vererbter Elementfunktionen

Durch das Hinzufügen neuer Elemente wird die Funktionalität der Basisklasse erweitert oder verändert. Sie können für Ihre abgeleitete Klasse einen neuen Konstruktor definieren, um die Konstruktoren der Basisklassen aufzurufen und alle neuen Datenelemente für die Klasse zu initialisieren.

Kombinieren von Objektverhalten

Wenn Sie mit ObjectWindows entwickeln und dabei mehrfache Vererbung verwenden, können Sie neue Klassen ableiten, die das Verhalten von mehr als einer Klasse erben. Solch ein »kombiniertes« Verhalten unterscheidet sich von dem aus einer einfachen Vererbung resultierenden Verhalten. Nicht das Verhalten der Basisklasse und dessen Erweiterung bzw. Änderung wird vererbt, sondern es wird das Verhalten mehrerer Klassen vererbt *und kombiniert*.

Wie bei der Ableitung mit einfacher Vererbung können Sie allerdings auch neue Elemente hinzufügen und vererbte überschreiben, um das Verhalten Ihrer neuen Klasse zu ändern.

Instantiierung von Klassen

- Um eine Klasse zu benutzen, müssen Sie eine Instanz dieser Klasse erzeugen. Sie können eine Klasse auf unterschiedliche Weise instantiieren:
- Sie können die Standardsyntax für Deklarationen verwenden. Die Syntax ist dieselbe wie bei der Deklaration irgendeiner Standardvariable wie **int** oder **char**. Im folgenden Beispiel wird *app* durch den Aufruf des Konstruktors *TMyApplication*, der keine Argumente besitzt, initialisiert:

  ```
  TMyApplication app;
  ```

  Diese Syntax kann nur verwendet werden, wenn die Klasse einen Standardkonstruktor oder einen Konstruktor, in dem alle Parameter Vorgabewerte haben, besitzt.

- Sie können die Standardsyntax auch für Deklarationen in Verbindung mit Argumenten verwenden, um einen speziellen Konstruktor aufzurufen. Im folgenden Beispiel wird *app* durch den Aufruf des Konstruktors *TMyApplication*, der **char*** als Argument besitzt, initialisiert:

  ```
  TMyApplication app("AppName");
  ```

- Sie können den Operator **new** verwenden, um Speicher für ein Objekt zu reservieren und dieses zu instantiieren. Beispiel:

```
TMyApplication *app;
app = new TMyApplication;
```

- Ebenso können Sie den Operator **new** zusammen mit Argumenten verwenden. Im folgenden Beispiel wird *app* durch den Aufruf des Konstruktors *TMyApplication*, der **char*** als Argument besitzt, initialisiert:

```
TMyApplication *app;
app = new TMyApplication("AppName");
```

  Die Konstruktoren rufen die Konstruktoren der Basisklassen auf und initialisieren alle benötigten Datenelemente. Sie können nur nicht-abstrakte Klassen instantiieren; das sind Klassen, die keine abstrakten Funktionen enthalten.

Abstrakte Klassen

Abstrakte Klassen enthalten abstrakte Funktionen, die Sie überschreiben müssen, um ihnen ein Verhalten vorzugeben. Die abstrakten Klassen dienen zwei Zwecken. Sie geben den konzeptuellen Rahmen vor, auf dem neue Klassen aufgebaut werden. Zum anderen reduzieren sie unter einem praktischen Aspekt den Programmieraufwand.

So könnten beispielsweise die ObjectWindows-Klassen *THSlider* und *TVSlider* jeweils direkt von *TScrollBar* abgeleitet werden. Obwohl eine horizontal und die andere vertikal ist, haben beide ähnliche Funktionalitäten und Reaktionen. Diese Gemeinsamkeit rechtfertigt die Erzeugung einer abstrakten Klasse mit dem Namen *TSlider*. *THSlider* und *TVSlider* werden dann von *TSlider* abgeleitet, wobei sie zusätzlich einige spezialisierte Elementfunktionen enthalten, um den Gleitregler unterschiedlich zu zeichnen.

Für abstrakte Klassen können keine Instanzen erzeugt werden. Die abstrakten Elementfunktionen müssen erst überschrieben werden, um eine sinnvolle Instanz zu erhalten. *TSlider* zum Beispiel weiß nicht, wie es selbst gezeichnet wird oder wie es auf Mausereignisse direkt reagieren soll.

Wenn Sie Ihren eigenen Gleitregler erzeugen wollen (zum Beispiel einen kreisförmigen Gleitregler), können Sie den Gleitregler von *TSlider* ableiten. Es kann aber einfacher sein, ihn von *TVSlider* oder von *THSlider* abzuleiten. Dies hängt von den jeweiligen Anforderungen ab. In jedem Falle fügen Sie Datenelemente und Elementfunktionen hinzu oder aber überschreiben Elementfunktionen, um die gewünschte Funktionalität zu erhalten. Wenn Sie Gleitregler für beide diagonale Richtungen haben wollen, könnten Sie eine abstrakte Zwischenklasse mit Namen *TAngledSlider* erzeugen.

Vererbung von Elementen

Das unten stehende Diagramm zeigt die Vererbung von *TInputDialog*. Danach wird *TInputDialog* von *TDialog* abgeleitet, *TDialog* von *TWindow* und *TWindow* wiederum von *TEventHandler* und *TStreamable*. Mit der Verer-

bung können Sie spezielleres Verhalten hinzufügen, während Sie sich in der Hierarchie vorwärts bewegen.

Abbildung 1.1: Die Vererbung bei *TDialog*

Die unten stehende Tabelle zeigt die **public** Datenelemente jeder Klasse einschließlich der Datenelemente, die von den Basisklassen geerbt sind.

Tabelle 1.1: Die Vererbung von Datenelementen

TWindow	TDialog	TInputDialog
Status	*Status*	*Status*
HWindow	*HWindow*	*HWindow*
Title	*Title*	*Title*
Parent	*Parent*	*Parent*
Attr	*Attr*	*Attr*
DefaultProc	*DefaultProc*	*DefaultProc*
Scroller	*Scroller*	*Scroller*
	IsModal	*IsModal*
		Prompt
		Buffer
		BufferSize

Die Klasse *TInputDialog* erbt alle Datenelemente von *TDialog* und *TWindow*. Zudem fügt sie alle weiteren Datenelemente hinzu, die für ein Eingabedialogfenster benötigt werden.

Um vollständig zu verstehen, wie *TInputDialog* verwendet werden kann, müssen sie dessen Vererbung verstehen: ein Objekt von *TInputDialog* ist sowohl ein Dialogfenster (*TDialog*) als auch ein Fenster (*TWindow*). *TDialog* fügt das Konzept der Modalität zur Klasse *TWindow* hinzu. *TInputDialog* erweitert dies durch die Fähigkeit, Daten aus Benutzereingaben zu speichern und wieder anzufordern.

Verschiedene Typen von Elementfunktionen

Es gibt vier (möglicherweise überlagerte) Typen von ObjectWindows-Elementfunktionen:

virtuell

- nichtvirtuell
- abstrakt

- Standardplatzhalter

Virtuelle Funktionen

Virtuelle Funktionen können in abgeleiteten Klassen überschrieben werden. Sie unterscheiden sich von abstrakten Funktionen darin, daß sie zur Verwendung der Klasse nicht unbedingt überschrieben werden *müssen*. Virtuelle Funktionen ermöglichen Ihnen den *Polymorphismus*, d.h. die Möglichkeit, eine konsistente Klassenschnittstelle bereitzustellen, auch wenn sich die Funktionalität Ihrer Klasse deutlich unterscheidet.

Nichtvirtuelle Funktionen

Sie sollten nichtvirtuelle Funktionen nicht überschreiben. Daher ist es wichtig, jede Elementfunktion als virtuell zu deklarieren, die in abgeleiteten Klassen überschrieben werden können (eine Ausnahme sind die Funktionen zur Behandlung von Ereignissen, die in Ihren Antworttabellen definiert sind). So ist beispielsweise *Twindow::CanClose* virtuell, da abgeleitete Klassen diese Funktion überschreiben sollten, um festzustellen, ob das Fenster schließen sollte. *TWindow::SetCaption* ist nichtvirtuell, da die Art und Weise, wie der Titel eines Fensters festgelegt wird, im allgemeinen nicht verändert werden muß.

Das Problem beim Überschreiben von nichtvirtuellen Funktionen besteht darin, daß Klassen, die von ihrer abgeleiteten Klasse abgeleitet werden, möglicherweise versuchen auf die überschriebene Funktion zuzugreifen. Wenn sich die neu abgeleiteten Klassen nicht *explizit* bewußt sind, daß Sie die Funktionalität der abgeleiteten Funktion geändert haben, kann dies zu fehlerhaften Rückgabewerten und Laufzeitfehlern führen.

Abstrakte Funktionen

Abstrakte Funktionen müssen in abgeleiteten Klassen überschrieben werden. Funktionen werden mit Hilfe des Initialisierers `= 0` als abstrakt markiert. Als Beispiel folgt die Deklaration von *TSlider::PaintRuler*:

```
virtual void PaintRuler(TDC& dc) = 0;
```

Sie müssen eine abstrakte Funktion nicht mit einer virtuellen Funktion überschreiben, wenn Sie nicht davon ausgehen, daß ein Überschreiben der Funktion in Zukunft nötig sein wird. Als Beispiel folgt die Deklaration von *THSlider::NotifyParent* , die die abstrakte Funktion *TSlider::NotifyParent* überschreibt:

```
void NotifyParent(int scrollCode, int pos = 0);
```

In einer abgeleiteten Klasse müssen Sie alle abstrakten Funktionen der abstrakten Klasse überschreiben, bevor Sie eine Instanz der abgeleiteten Klasse erzeugen können. Wenn Sie die Standardklassen von ObjectWindows verwenden, wird dies in den meisten Fällen kein besonderes Problem darstellen. Die meisten ObjectWindows-Klassen, die Sie für eine Ableitung benötigen, sind nichtabstrakte Klassen. Viele ObjectWindows-

Klassen verwenden anstatt der abstrakten Funktionen einfach die Standardplatzhalter-Funktionen.

Standardplatzhalter-Funktionen

Verglichen mit abstrakten Funktionen müssen Standardplatzhalter-Funktionen nicht überschrieben werden. Sie bieten minimale Standardbefehle bzw. überhaupt keine Befehle. Sie dienen als Platzhalter für Code, den Sie in Ihre abgeleitete Klasse einfügen können. Hier ist beispielsweise die Definition von *TWindow::EvLButtonDblClk*:

```
inline void TWindow::EvLButtonDblClk (UINT modKeys, TPoint &) {
    DefaultProcessing();
    }
```

EvLButtonDblClk ruft standardmäßig einfach *DefaultProcessing* auf, um die Standardverarbeitung für diese Botschaft auszuführen. In Ihrer eigenen Fensterklasse können Sie *EvLButtonDblClk* überschreiben, indem Sie die Funktion in der Antworttabelle Ihrer Klasse definieren. Ihre Version von *EvLButtonDblClk* kann dann ein spezielles Verhalten bereitstellen, das der vom Anwender gewünschten Reaktion auf das Drücken der linken Maustaste entspricht. Ebenso können Sie weiterhin die Standardverarbeitung der Basisklasse bereitstellen, indem Sie die Funktionsversion der Basisklasse aufrufen.

Objekttypologie

Die Hierarchie von ObjectWindows enthält viele verschiedene Typen von Klassen, die Sie benutzen, verändern oder erweitern können. Die Aufgaben jeder Klasse können nach den folgenden Gruppen eingeteilt werden:

- Fenster
- Dialogfenster
- Dialogelemente
- Grafiken
- Drucken
- Module und Anwendungen
- Dok/View-Anwendungen
- Verschiedene Windows-Elemente

Fensterklassen

Ein wichtiger Bestandteil jeder Windows-Anwendung ist natürlich das Fenster. ObjectWindows bietet mehrere verschiedene Fensterklassen für verschiedene Fenstertypen (nicht zu verwechseln mit den Registriertypen von Windows für »Fensterklassen«):

- Fenster
- Rahmenfenster

- MDI-Fenster
- Dekorierte Fenster

In Kapitel 8 werden die Fensterklassen im Detail erklärt.

Fenster

TWindow ist die Basisklasse für alle Fensterklassen. Sie repräsentiert die für alle Fenster gemeinsame Funktionalität, ungeachtet der Tatsache, ob eine Fensterklasse ein Dialogfenster, ein Dialogelement, ein MDI-Fenster usw.

Rahmenfenster

TFrameWindow ist von *TWindow* abgeleitet und fügt die Funktionalität eines Rahmenfensters hinzu, welches andere Client-Fenster enthalten kann.

MDI-Fenster

Das »Multiple Document Interface« (MDI) ist in Windows der Standard für den Umgang mit mehreren Dokumenten oder Fenstern in einer einzigen Anwendung. *TMDIFrame*, *TMDIClient*, und *TMDIChild* unterstützen MDI in ObjectWindows-Anwendungen.

Dekorierte Fenster

Verschiedene Klassen, wie beispielsweise *TLayoutWindow* und *TLayoutMetrics*, werden in Kombination benutzt, um die *Dekoration* von Dialogelementen wie Tool-Balken, Statuszeilen und Meldungszeilen zu unterstützen. Durch mehrfache Vererbung wird die Möglichkeit zur Dekoration auch in Rahmenfenstern und MDI-Rahmenfenstern in *TDecoratedFrame* und *TDecoratedMDIFrame* hinzugefügt.

Dialogfensterklassen

Die Klasse *TDialog* ist von *TWindow* abgeleitet.

Sie wird für die Erzeugung von Dialogfenstern verwendet, die eine Reihe von Benutzereingaben handhaben. Dialogfenster enthalten üblicherweise Dialogelemente, um die Eingaben von Benutzern aufzunehmen. Die Klassen von Dialogfenstern werden in Kapitel 10 genauer beschrieben.

Allgemeine Dialogfenster

Zusätzlich zu den für Ihre Anwendung besonders eingerichteten Dialogfenstern unterstützt ObjectWindows auch allgemeine Dialogfenster von Windows für folgende Zwecke:

- Auswahl von Dateien (*TOpenSaveDialog*, *TFileOpenDialog* und *TFileSaveDialog*)
- Auswahl von Schriftarten (*TChooseFontDialog*)
- Auswahl von Farben (*TChooseColorDialog*)
- Auswahl von Druckoptionen (*TPrintDialog*)
- Suchen und Ersetzen von Text (*TFindReplaceDialog*, *TFindDialog* und *TReplaceDialog*)

Andere Dialoge

ObjectWindows bietet ebenso einige andere Dialogfenster, die nicht auf den allgemeinen Dialogfenstern von Windows basieren:

- Eingabe von Text (*TInputDialog*)
- Abbruch von Druckaufträgen (*TPrinterAbortDlg*, in Verbindung mit den Klassen *TPrinter* und *TPrintout* verwendet)

Klassen für Dialogelemente

TControl ist eine von *TWindow* abgeleitete Klasse, um Verhalten bereitzustellen, das allen Dialogelementen gemeinsam ist. ObjectWindows kennt drei Typen von Dialogelementen

- Standarddialogelemente von Windows
- Hilfselemente (widgets)
- Dekorationen

Alle diese Dialogelemente werden in Kapitel 12 genauer beschrieben.

Standarddialogelemente von Windows

Standarddialogelemente von Windows umfassen Listenfelder, Bildlaufleisten, Schalter, Markierungsfelder, Schaltfelder, Gruppenfelder, Editierfelder, statische Elemente und Kombinationsfenster. Mit Hilfe von Elementfunktionen können diese Dialogelemente manipuliert werden.

Hilfselemente

Im Gegensatz zu den Standarddialogelementen von Windows sind die Hilfselemente von ObjectWindows spezialisierte Dialogelemente, die vollständig in C++ geschrieben sind. Die in ObjectWindows angebotenen Hilfselemente bieten horizontale und vertikale Gleitregler (*THSlider* und *TVSlider*) und Meßanzeigen (*TGauge*).

Dekorationen

Dekorationen sind spezialisierte untergeordnete Fenster, die dem Benutzer erstens die Auswahl eines Kommandos ermöglichen, zweitens eine Stelle bieten, um dem Benutzer Informationen zu geben, und drittens eine spezialisierte Kommunikation mit dem Benutzer zulassen. Eine Dialogelementleiste (*TControlBar*) zum Beispiel erlaubt die Anordnung einer Anzahl von Schaltern auf einer mit dem Fenster verbundenen Leiste als Abkürzung für die Benutzung von Menüs. Eine Toolbox (*TToolbox*) erlaubt die Anordnung von Schaltern auf einer fließenden Palette. Meldungszeilen (*TMessageBar*) sind Zeilen, die im allgemeinen am unteren Rand eines Fensters erscheinen und über die Informationen für den Benutzer angezeigt werden können. Statuszeilen (*TStatusBar*) haben Ähnlichkeit mit den Meldungszeilen, sie bieten allerdings Raum für mehr als eine Information. Die Statuszeile in der IDE von Borland C++ zeigt Ihre Position im Editierfenster, den Bearbeitungsmodus (Einfügen oder Überschreiben) und Fehlermeldungen an.

Klassen für Grafik

Windows bietet eine mächtige, aber auch komplexe Grafikbibliothek, die als »grafische Geräteschnittstelle« (GDI) bezeichnet wird. ObjectWindows kapselt die GDI, um ihre Verwendung in der Anwendung einfacher zu gestalten:

- Klassen für den Gerätekontext (auch DC) (*TDC*)
- GDI-Objekte (*TGDIObject*)

Siehe das Kapitel 15 für eine genaue Beschreibung dieser Klassen.

DC-Klassen

Mit der GDI zeichnen Sie nicht direkt auf ein Gerät (wie Bildschirm oder Drucker), sondern auf ein Bitmap unter Verwendung eines Gerätekontextes (DC). Ein *Gerätekontext* ist eine Sammlung von Werkzeugen, Einstellungen und Geräteinformationen bezogen auf ein grafisches Gerät und seinen aktuellen Darstellungszustand. Über die Verwendung von GDI-Funktionen wird damit ein hoher Grad an Geräteunabhängigkeit möglich. Die folgende Tabelle listet die unterschiedlichen Typen von DCs auf, die ObjectWindows kapselt.

Tabelle 1.2: Gerätekontexte, die von ObjectWindows gekapselt werden

Gerätekontexttyp	ObjectWindows-DC-Klasse
Speicher	*TMemoryDC*
Metadatei	*TMetaFileDC*
Bitmap	*TDibDC*
Drucker	*TPrintDC*
Fenster	*TWindowDC*
Desktop	*TDesktopDC*
Bildschirm	*TScreenDC*
Client	*TClientDC*
Paint	*TPaintDC*

GDI-Objekte

TGDIObject ist eine Basisklasse für mehrere andere Klassen, die Hilfsmittel zum Zeichnen und zur Kontrolle von Zeichnungen repräsentieren.

Die folgende Tabelle listet diese Klassen und andere GDI-Hilfsklassen von ObjectWindows auf.

Tabelle 1.3: GDI-Hilfsklassen

GDI-Objekttyp	ObjectWindows-GDI-Klasse
Stifte	*TPen*
Pinsel	*TBrush*
Schriftarten	*TFont*
Paletten	*TPalette*
Bitmaps	*TBitmap*, *TDib*, *TUIBitmap*
Symbole	*TIcon*
Cursor	*TCursor*
Regionen	*TRegion*
Punkte	*TPoint*

Größe	*TSize*
Rechtecke	*TRect*
Farbspezifizierer	*TColor*
Dreifarben-RGB	*TRgbTriple*
Vierfarben-RGB	*TRgbQuad*
Paletteneinträge	*TPaletteEntry*
Metadatei	*TMetafilePict*

Klassen zum Drucken

TPrinter vereinfacht das Drucken bedeutend, indem sie die Kommunikation mit den Druckertreibern kapselt. *TPrintout* kapselt die Befehle für den Ausdruck eines Dokumentes. In Kapitel 14 wird der Gebrauch von Klassen zum Drucken erläutert.

Klassen für Module und Anwendungen

Eine Windows-Anwendung ist dafür verantwortlich, daß Fenster initialisiert werden und daß die zu ihr von Windows geschickten Botschaften auch zum entsprechenden Fenster weitergereicht werden. ObjectWindows kapselt dieses Verhalten in *TApplication*. Das Verhalten einer DLL ist in *TDll* enthalten. Siehe Kapitel 3 für ausführliche Details über Modul- und Anwendungsobjekte.

Dok/View-Klassen

Die Klassen zur Darstellung von Dokumenten sind eine vollständige Abstraktion eines generischen Modells für ein Dokument und dessen Darstellung. Die Basisklassen des Dok/View-Modells sind *TDocManager*, *TDocument* und *TView*. Das Dok/View-Modell stellt ein System dar, in dem Daten enthalten sind, in dem auf diese über ein Dokumentobjekt zugegriffen wird und in dem die Daten über ein View-Objekt manipuliert werden. Eine beliebige Anzahl von Views kann mit einem einzelnen Dokumenttyp assoziiert werden. Damit können Sie dieselben Daten auf eine Reihe von unterschiedlichen Arten darstellen.

Sie können zum Beispiel eine Linie grafisch als eine Linie in einem Fenster wie auch als Punktmenge darstellen, die die Koordinaten der Eckpunkte der Linie angibt. Dies würde ein einziges Dokument erfordern, das die Daten sowie zwei View-Klassen enthält, von denen eine die Linie auf dem Bildschirm anzeigt und die andere die Koordinaten der Punkte in der Linie. Sie können zudem die Daten über die Views verändern. So können Sie in diesem Beispiel die Daten der Linie entweder durch Zeichnen in der grafischen Anzeige oder durch Eingabe von Zahlen, Verändern und Hinzufügen von Koordinaten in der numerischen Anzeige manipulieren.

Das Dok/View-Modell wird in Kapitel 11 genau beschrieben.

Verschiedene Klassen

Da Windows sehr vielseitig ist, passen nicht alle von ObjectWindows bereitgestellten Klassen in klar einordnare Kategorien. Dieser Abschnitt beschreibt diese verschiedenen Klassen.

Menüs

Menüs können statisch sein, Sie können sie aber ebenso verändern oder auch vollständige, neue Menüs laden. *TMenu* und ihre abgeleiteten Klassen (*TSystemMenu* und *TPopupMenu*) ermöglicht eine leichte Manipulation von Menüs. In Kapitel 9 werden die Menüklassen eingehender erläutert.

Zwischenablage

Die Zwischenablage von Windows ist eine der Hauptmöglichkeiten, über die der Benutzer Daten zwischen Anwendungen austauschen kann. Ein *TClipboard*-Objekt von ObjectWindows ermöglicht den einfachen Umgang mit der Zwischenablage in Ihren Anwendungen. Für weitere Details siehe Kapitel 19.

K A P I T E L 2

Eine Einführung in ObjectWindows

Die Einführung in ObjectWindows 2.5 beschreibt die Grundlagen für die Programmierung für Windows mit Hilfe des Anwendungswerkzeuges ObjectWindows. Die Einführung enthält eine Anwendung, die in zwölf immer komplizierter werdenden ObjectWindows-Schritten entwickelt wird. Jeder weitere Schritt in der Anwendung repräsentiert auch einen weiteren Schritt bei den Lektionen der Einführung. Nach dem zwölften Schritt werden wir eine voll funktionsfähige Windows-Anwendung fertiggestellt haben, die solche Merkmale wie Menüs, Dialogfenster, grafische Dialogelementleisten, Statuszeilen, MDI-Fenster und weiteres mehr aufweisen wird.

Diese Einführung geht davon aus, daß Sie mit der Sprache C++ vertraut sind und bereits einige Erfahrung mit der Windows-Programmierung besitzen. Bevor Sie diese Einführung beginnen, könnte es für Sie hilfreich sein, Kapitel 1 zu lesen. Dieses Kapitel enthält eine kurze, informelle Übersicht über die Klassenhierarchie von ObjectWindows 2.5. Dies könnte für Sie die Prinzipien, die hinter der Struktur der Klassenbibliothek von ObjectWindows stehen, durchschaubarer machen.

Die Einführung erläutert jede neue Version der Anwendung sowie die Unterschiede zwischen ihr und der vorherigen Version. Jede Erklärung beinhaltet zudem Hinweise auf mögliche Anwendungen der aktuellen Lektion in verschiedenen realen Problemstellungen. Am Ende jeder Lektion befindet sich ein Referenzabschnitt, der auf die Stellen verweist, die weitere Information zu den in diesem Abschnitt behandelten Themen enthält.

Erste Schritte

Bevor Sie diese Einführung beginnen, sollten Sie eine Kopie der Dateien für die ObjectWindows-Einführung separat von den Dateien in Ihrer

Compiler-Installation anlegen. Verwenden Sie die kopierten Dateien, wenn Sie die einzelnen Schritte der Einführung durcharbeiten. In den einzelnen Schritten sollten Sie die Änderungen am Programm selbst versuchen. Sie können dann die von Ihnen vorgenommenen Änderungen mit dem Programm der Einführung jeweils vergleichen.

Dateien für die Einführung

Die Einführung ist aus einer Reihe von Quelltextdateien zusammengesetzt:

- Jeder Schritt der Einführung ist in einer Datei mit Namen STEP*XX*.CPP enthalten.
- Spätere Schritte für die Anwendung verwenden mehrere C++ Quelltextdateien. Die anderen Dateien haben den Namen STEP*XX*DV.CPP.
- Eine Reihe von Schritten besitzt eine Header-Datei, die Klassendefinitionen und Ähnliches enthält. Diese Header-Dateien haben den Namen STEP*XX*DV.H.
- Eine Reihe von Schritten besitzt ferner eine korrespondierende Ressourcen-Script-Datei mit Namen STEP*XX*.RC.

Bei all diesen Fällen ist *XX* eine Zahl von »01« bis »12«, die anzeigt, welcher Schritt der Einführung sich in der Quelltextdatei befindet.

Schritt 1: Die grundlegende Anwendung

Öffnen Sie zu Beginn der Einführung die Datei STEP01.CPP. Sie zeigt ein Beispiel einer minimalen, sinnvollen ObjectWindows-Anwendung. Wegen ihrer Kürze kann die komplette Datei hier gezeigt werden:

Der Quelltext für Schritt 1 befindet sich in der Datei STEP01.CPP im Verzeichnis EXAMPLES\OWL\TUTORIAL.

```
//----------------------------------------------------------------------------
// ObjectWindows - (C) Copyright 1991, 1993 von Borland International
//   Beispielanwendung -- step01.cpp
//----------------------------------------------------------------------------
#include <owl\applicat.h>
#include <owl\framewin.h>

class TMyApp : public TApplication
{
  public:
    TMyApp() : TApplication() {}


    void InitMainWindow()
    {
      SetMainWindow(new TFrameWindow(0, "Sample ObjectWindows
                    Program"));
```

```
    }
};


int OwlMain(int /*argc*/, char* /*argv*/ [])
{
  return TMyApp().Run();
}
```

Diese einfache Anwendung enthält eine Reihe wichtiger Merkmale:

- Diese Quelltextdatei bindet zwei Header-Dateien mit ein, owl\applicat.h und owl\framewin.h. Diese Dateien sind mit eingebunden, da die Anwendung die ObjectWindows-Klassen *TApplication* und *TFrameWindow* benutzt. Immer wenn Sie eine ObjectWindows-Klasse benutzen, müssen Sie die geeigneten Header-Dateien mit einbinden, damit Ihr Code korrekt compiliert werden kann.
- Die Klasse *TMyApp* ist von der ObjectWindows-Klasse *TApplication* abgeleitet. Jede ObjectWindows-Anwendung besitzt ein *TApplication*-Objekt – oder eher noch ein von *TApplication* abgeleitetes Objekt –, das als Anwendungsobjekt bezeichnet wird. Wenn Sie ein *TApplication*-Objekt direkt verwenden wollen, können Sie Schwierigkeiten bei der Kontrolle des Programmflusses bekommen. Ein Überschreiben von *TApplication* ermöglicht Ihnen einen Zugriff auf die Arbeitsweise des Anwendungsobjektes wie auch ein Ändern der Funktionen, die für die von Ihnen gewünschte Arbeitsweise der Anwendung überschrieben werden müssen.
- Zusätzlich zu einem Anwendungsobjekt besitzt jede ObjectWindows-Anwendung eine Funktion *OwlMain*. Das Anwendungsobjekt wird tatsächlich in der Funktion *OwlMain* mit einer einfachen Deklaration erzeugt. *OwlMain* ist das ObjectWindows-Äquivalent der Funktion *WinMain* in einer regulären *Windows*-Anwendung. Sie können *OwlMain* verwenden, um Kommandozeilenargumente zu überprüfen, globale Daten einzurichten oder um alles andere durchzuführen, was vor dem Start der Anwendungsausführung erledigt sein muß.
- Um die Ausführung der Anwendung zu starten, rufen Sie die Funktion *Run* des Anwendungsobjektes auf. Die Funktion *Run* ruft zuerst die Funktion *InitApplication* auf, allerdings nur, sofern es sich hierbei um die erste Instanz der Anwendung handelt (die vorgegebene Funktion *TApplication::InitApplication* hat keinen Effekt). Nach der Rückkehr der Funktion *InitApplication* ruft *Run* die Funktion *InitInstance* auf, welche jede Instanz einer Anwendung initialisiert. Die vorgegebene Funktion *TApplication::InitInstance* ruft die Funktion *InitMainWindow* auf, welche

das Hauptfenster der Anwendung initialisiert, sie erzeugt dann das Hauptfenster und zeigt es auf dem Bildschirm an.

- *TMyApp* überschreibt die Funktion *InitMainWindow*. Sie können diese Funktion verwenden, um das Hauptfenster nach Ihren Wünschen zu gestalten. Die Funktion *SetMainWindow* setzt das Hauptfenster der Anwendung auf ein *TFrameWindow* oder ein von *TFrameWindow* abgeleitetes Objekt, das an die Funktion übergeben wird. In diesem Falle erzeugen Sie einfach ein neues *TFrameWindow* ohne übergeordnetes Fenster (der erste Parameter von *TFrameWindow* ist ein Zeiger auf das zum Fenster übergeordnete Fenster) und dem Titel »Sample ObjectWindows Program«.

Diese grundlegende Anwendung führt zwei der wichtigsten Konzepte bei der ObjectWindows-Programmierung ein. So einfach es auch erscheint, das Ableiten einer Klasse von *TApplication* und das Überschreiben der Funktion *InitMainWindow* ermöglicht Ihnen eine weitreichende Kontrolle über die Ausführung der Anwendung. Wie Sie in späteren Schritten sehen werden, können Sie eine große und komplexe Anwendung leicht von dieser einfachen Anwendung aus beginnend aufbauen.

Weiterführende Informationen

Es folgt eine Aufstellung davon, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Anwendungsobjekte werden zusammen mit deren *Init**-Elementfunktionen in Kapitel 3 beschrieben.
- *OwlMain* wird in Kapitel 3 beschrieben.
- *TFrameWindow* wird in Kapitel 15 beschrieben.

Schritt 2: Behandlung von Windows-Ereignissen

Sie können den Quelltext für Schritt 2 in der Datei STEP02.CPP im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

Schritt 2 führt Antworttabellen ein, die ein weiteres wichtiges Merkmal von ObjectWindows darstellen. Antworttabellen kontrollieren die Verarbeitung von Ereignissen und Botschaften in ObjectWindows 2.5-Anwendungen, indem sie Ereignisse den geeigneten Ereignisbehandlungsfunktionen zuteilen. Schritt 2 fügt ebenso diese Funktionen hinzu.

Hinzufügen einer Fensterklasse

Fügen Sie die Antworttabelle mit Hilfe einer Fensterklasse mit Namen *TMyWindow* der Anwendung hinzu. *TMyWindow* ist von *TWindow* abgeleitet und sieht wie folgt aus:

```
class TMyWindow : public TWindow
{
  public:
    TMyWindow(TWindow* parent = 0);
```

```
  protected:
    // Überschreibe Elementfunktion von TWindow
    BOOL CanClose();

    // Botschaftsantwortfunktionen
    void EvLButtonDown(UINT, TPoint&);
    void EvRButtonDown(UINT, TPoint&);

    DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

Der Konstruktor für diese Klasse ist sehr einfach. Er erwartet ein *TWindow* * als einzigen Parameter, der das dem Objekt übergeordnete Fenster anzeigt. Die Konstruktordefinition sieht wie folgt aus:

```
TMyWindow::TMyWindow(TWindow *parent)
{
  Init(parent, 0, 0);
}
```

Die Funktion *Init* ermöglicht Ihnen die Initialisierung der Basisklasse von *TMyWindow*. In diesem Fall ist der Aufruf nicht sehr kompliziert. Die einzige für Ihre Zwecke eventuell erforderliche Information ist das dem Fenster übergeordnete Fenster und Sie werden sehen, daß auch dies für Sie übernommen wird.

Hinzufügen einer Antworttabelle

Das einzige öffentliche Element der Klasse *TMyWindow* ist ihr Konstruktor. Wenn aber die anderen Elemente **protected** sind, wie können Sie dann auf sie zugreifen?. Die Antwort liegt in der Definition der Antworttabelle. Beachten Sie die letze Zeile der Klassendefinition von *TMyWindow*. Diese deklariert die Antworttabelle; das heißt, dies informiert Ihre Klasse, daß sie eine Antworttabelle besitzt. Dies ist vergleichbar mit einer Funktionsdeklaration, welche die Klasse über die Existenz der Funktion informiert, aber nicht die Aktivitäten der Funktion definiert.

Die Definition der Antworttabelle richtet Ihre Klasse ein, damit sie Windows-Ereignisse behandeln und jedes Ereignis an die geeignete Ereignisbehandlungsfunktion übergeben kann. Nach einer generellen Regel sollten Ereignisbehandlungsfunktionen **protected** sein. Dadurch wird verhindert, daß Klassen und Funktionen außerhalb Ihrer eigenen Klasse diese Behandlungsfunktionen aufrufen. Hier ist die Definition der Antworttabelle für *TMyWindow*:

```
DEFINE_RESPONSE_TABLE1(TMyWindow, TWindow)
  EV_WM_LBUTTONDOWN,
  EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
```

Sie können die Antworttabelle irgendwo in Ihrer Quelltextdatei ablegen.

Für die hier verfolgten Zwecke können Sie die Antworttabelle einfach halten. Hier ist eine Beschreibung von jedem Teil der Tabelle. Eine Antworttabelle besitzt vier wichtige Bestandteile:

- Die Deklaration der Antworttabelle in der Klassendeklaration.
- Die erste Zeile einer Antworttabellendefinition ist immer das Makro DEFINE_RESPONSE_TABLE*X*. Der Wert von *X* hängt von der Vererbung Ihrer Klasse ab und basiert auf der Zahl der unmittelbaren Basisklassen für Ihre Klasse. In diesem Fall besitzt *TMyWindow* nur eine unmittelbare Basisklasse, *TWindow*.
- Die letzte Zeile einer Antworttabellendefinition ist immer das Makro END_RESPONSE_TABLE, welches die Definition der Ereignisantworttabelle beendet.
- Zwischen dem Makro DEFINE_RESPONSE_TABLE*X* und dem Makro END_RESPONSE_TABLE befinden sich weitere Makros, die einzelne Ereignisse mit deren Behandlungsfunktionen assoziieren.

Die beiden Makros EV_WM_LBUTTONDOWN und EV_WM_RBUTTONDOWN in der Mitte der Antworttabelle sind Antworttabellenmakros für die Windows-Standardbotschaften WM_LBUTTONDOWN und WM_RBUTTONDOWN. Alle Windows-Standardbotschaften besitzen von ObjectWindows definierte Antworttabellenmakros. Um den Namen eines einzelnen Makros einer Botschaft zu ermitteln, müssen Sie dem Botschaftsnamen nur ein »EV_« voranstellen. So hat zum Beispiel das Makro, welches die Botschaft WM_PAINT behandelt, den Namen EV_WM_PAINT und das Makro für die Botschaft WM_LBUTTONDOWN heißt EV_WM_LBUTTONDOWN.

Diese vordefinierten Makros übergeben die Botschaft an Funktionen mit vordefiniertem Namen. Um den Funktionsnamen zu ermitteln, müssen Sie *Ev* für »WM_« einsetzen und in Kleinbuchstaben konvertieren, wobei die einzelnen Worte im Namen mit Großbuchstaben beginnen. So wird zum Beispiel die Botschaft WM_PAINT an eine Funktion mit Namen *EvPaint* übergeben und die Botschaft WM_LBUTTONDOWN ein eine Funktion mit Namen *EvLButtonDown*.

Ereignisbehandlungsfunktionen

Wie Sie sehen können, sind zwei der Funktionen mit Zugriffsrecht **protected** in *TMyWindow* die Funktionen *EvLButtonDown* und *EvRButtonDown*. Dank der Makros in der Antworttabelle kann *TMyWindow* bei Empfang eines WM_LBUTTONDOWN- oder WM_RBUTTONDOWN-Ereignisses dieses an die geeignete Funktion weiterreichen.

Die Funktionen, die die Ereignisse WM_LBUTTONDOWN oder WM_RBUTTONDOWN behandeln, sind sehr einfach. Jede Funktion zeigt ein Meldungsfenster, welches Ihnen mitteilt, welchen Schalter Sie

gedrückt haben. Der Code für diese Funktionen sollte in etwa wie folgt aussehen:

```
void TMyWindow::EvLButtonDown(UINT, TPoint&)
{
  MessageBox("You have pressed the left mouse button",
             "Message Dispatched", MB_OK);
}

void TMyWindow::EvRButtonDown(UINT, TPoint&)
{
  MessageBox("You have pressed the right mouse button",
             "Message Dispatched", MB_OK);
}
```

Dies verdeutlicht eines der besten Merkmale von ObjectWindows 2.5 zur Behandlung von Windows-Standardereignissen. Die Funktion, die ein Ereignis behandelt, empfängt einen scheinbar fast beliebigen Parametertyp (Alle Makros und deren korrespondierende Funktionen sind im ObjectWindows *Referenzhandbuch* aufgeführt). In Wirklichkeit korrespondieren diese Parametertypen zur Information, die in den Variablen WPARAM und LPARAM codiert ist, welche normalerweise zusammen mit dem Ereignis übergeben werden. Diese Ereignisinformation wird für Sie automatisch »extrahiert«.

Dieser Ansatz zeigt zwei Vorteile:

- Sie müssen nicht mehr die Information aus den Werten WPARAM und LPARAM manuell extrahieren.
- Die vordefinierten Funktionen erlauben eine Typüberprüfung zur Compilierzeit und verhindern schwer nachvollziehbare Fehler, die durch ein Verwechseln der in WPARAM und LPARAM codierten Werte hervorgerufen werden können.

So enthalten zum Beispiel sowohl WM_LBUTTONDOWN als auch WM_RBUTTONDOWN denselben Typ von Information in ihren Variablen WPARAM und LPARAM:

- WPARAM enthält Tasten-Flags, die angeben, ob der Benutzer eine Taste aus einer Reihe virtueller Tasten gedrückt hat.
- Das niederwertige Wort von LPARAM gibt die x-Koordinate des Mauszeigers an.
- Das höherwertige Wort von LPARAM gibt die y-Koordinate des Mauszeigers an.

EvLButtonDown und *EvRButtonDown* besitzen auch einen ähnlichen Funktionskopf. Der UINT-Parameter jeder Funktion korrespondiert mit dem Tasten-Flags-Parameter. Die Werte, die normalerweise in LPARAM codiert sind, sind hier jedoch in einem *TPoint*-Objekt gespeichert.

Gekapselte API-Aufrufe

Vielleicht ist Ihnen aufgefallen, daß die Aufrufe der Funktion *MessageBox* etwas merkwürdig aussehen. Die Windows-API-Funktion *MessageBox* erwartet ein HWND als ersten Parameter. Die hier aufgerufene Funktion *MessageBox* ist jedoch in Wirklichkeit eine Elementfunktion der Klasse *TWindow*. Es gibt eine große Zahl von Funktionen, die denselben Namen wie die Windows-API-Funktion haben, deren Funktionskopf sich jedoch unterscheidet. Die häufigsten Unterscheidungsmerkmale sind der Wegfall der Handle-Parameter wie zum Beispiel HWND und HINSTANCE, die Ersetzung von Windows-Datentypen durch ObjectWindows-Datentypen usw. In diesem Fall stellt die Fensterklasse den Parameter HWND für Sie bereit.

Überschreiben der Funktion CanClose

Ein anderes Merkmal der Klasse *TMyWindow* ist die Funktion *CanClose*. Bevor eine Anwendung versucht, ein Fenster zu schließen, ruft sie die Funktion *CanClose* des Fensters auf. Das Fenster kann das Schließen abbrechen, indem FALSE zurückgegeben wird, oder das Schließen zulassen, indem TRUE zurückgegeben wird.

Aus der Sicht der Anwendung wird damit sichergestellt, daß Sie kein Fensters schließen, das aktuell benutzt wird, oder das noch nicht gespeicherte Daten enthält. Aus der Sicht des Fensters erhalten Sie eine Warnung, wenn die Anwendung ein Schließen versucht und bietet Ihnen die Möglichkeit, alles nötige vor dem Schließen aufzuräumen. Hier ist die Funktion *CanClose* der Klasse *TMyWindow*:

```
BOOL TMyWindow::CanClose()
{
  return MessageBox("Do you want to save?", "Drawing has changed",
                    MB_YESNO | MB_ICONQUESTION) == IDNO;
}
```

Für den hier verfolgten Zweck wirft diese Funktion nur ein Meldungsfenster auf, welches mitteilt, daß sich die Darstellung geändert hat, und nachfragt, ob der Benutzer die Darstellung speichern will. Da es keine zu speichernde Darstellung gibt, ist diese Meldung nicht gerade sinnvoll. Sie wird es jedoch in Schritt 7, bei dem Sie die Möglichkeit hinzufügen, Daten in eine Datei zu speichern.

Verwenden von TMyWindow als Hauptfenster

Als letztes muß eine Instanz dieser neuen Klasse *TMyWindow* erzeugt werden. Sie könnten nun glauben, Sie müßten dazu einfach im Aufruf von *SetMainWindow* in der Funktion *InitMainWindow* die Klasse *TFrameWindow* durch *TMyWindow* ersetzen:

```
void InitMainWindow()
{
  SetMainWindow(new TMyWindow);
}
```

Aus einer Reihe von Gründen wird dies nicht funktionieren. Ein primärer Grund dafür liegt jedoch darin, daß *TMyWindow* nicht auf *TFrameWindow* basiert. Damit dieser Code korrekt compiliert werden kann, müßten Sie *TMyWindow* ändern, so daß sie auf *TFrameWindow* anstatt auf *TWindow* basiert. Auch wenn dies auf einfache Weise geschehen kann, führt dies für die Klasse *TMyWindow* neue Funktionalität ein, die nicht notwendig ist. Wie Sie in späteren Schritten sehen werden, dient *TMyWindow* einem eindeutigen Zweck. Wenn man die Fähigkeiten eines Rahmenfensters der Klasse *TMyWindow* hinzufügen würde, würde ihre Flexibilität reduziert werden.

Beim zweiten Ansatz verwendet man ein *TMyWindow*-Objekt als ein Client in einem *TFrameWindow*. Dies ist eine leichte Aufgabe: der dritte Parameter der Konstruktors von *TFrameWindow*, den Sie bereits benutzen, erlaubt Ihnen die Angabe eines *TWindow*-Objektes bzw. eines von *TWindow* abgeleiteten Objektes als ein Client zum Rahmenfenster. Der Code würde in etwa wie folgt aussehen:

```
SetMainWindow(new TFrameWindow(0, "Sample ObjectWindows Program",
                               new TMyWindow));
```

Bei diesem Ansatz verwaltet *TFrameWindow* das Rahmenfenster und befreit *TMyWindow* für die Erfüllung ihrer eigenen Aufgaben. Dies fördert den diskreten und modularen Objektentwurf. Wie Sie in Schritt 10 sehen werden, ermöglicht es Ihnen zudem eine einfache Änderung des von Ihnen verwendeten Rahmenfenstertyps.

Beachten Sie, daß die Konstruktion **new** *TMyWindow* im Konstruktor von *TFrameWindow* kein übergeordnetes Fenster für das *TMyWindow*-Objekt angibt. Der Grund dafür liegt darin, daß noch nichts für ein übergeordnetes Fenster existiert. Das *TFrameWindow*-Objekt, welches das übergeordnete Fenster darstellen wird, wurde noch nicht konstruiert. *TFrameWindow* setzt automatisch das zum Client-Fenster übergeordnete Fenster auf ein *TFrameWindow*, sobald des konstruiert wurde.

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Fensterklassen sind in Kapitel 8 beschrieben.
- Oberflächenobjekte im allgemeinen, wie zum Beispiel Fenster, Dialogfenster, Dialogelemente usw., sind in Kapitel 4 beschrieben.
- Antworttabellen sind in Kapitel 5 beschrieben.
- Hauptfenster sind in Kapitel 3 beschrieben.
- Vordefinierte Antworttabellenmakros und deren korrespondierenden Ereignisbehandlungsfunktionen sind in Kapitel 3 im ObjectWindows *Referenzhandbuch* aufgelistet.

Schritt 3: Schreiben in das Fenster

In Schritt 3 werden Sie mit dem neuen Fenster, das in Schritt 2 Ihrer Anwendung hinzugefügt wurde, arbeiten. Anstatt ein Meldungsfenster anzuzeigen, wenn die Maustaste gedrückt wurde, werden die Ereignisbehandlungsfunktionen eine echte Funktionalität erhalten – beim Drücken der linken Maustaste werden die Koordinaten des Punktes, an dem die Taste gedrückt wurde, in das Fenster gedruckt und beim Drücken der rechten Maustaste wird das Fenster geleert.

Sie können den Quelltext für Schritt 3 in der Datei STEP03.CPP im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

Der Code für diese neue Funktionalität liegt in der Funktion *EvLButtonDown*. Der Parameter *TPoint*, der an *EvLButtonDown* übergeben wird, enthält die Koordinaten des Punktes, an dem die Maustaste gedrückt wurde. Sie werden ein **char**-String der Funktion hinzufügen müssen, damit die Textrepräsentation des Punktes aufgenommen werden kann. Sie können dann die Funktion *wsprintf* benutzen, um den String zu formatieren. Nun müssen Sie das Fenster einrichten, um den String zu drucken.

Konstruieren eines Gerätekontextes

Um jede Art grafischer Operationen in Windows durchzuführen, müssen Sie einen Gerätekontext für das Fenster oder den Bereich, mit dem Sie arbeiten wollen, haben. Dasselbe trifft auch für ObjectWindows zu. ObjectWindows bietet eine Reihe von Klassen, die das Aufstellen, Benutzen und Einrichten eines Gerätekontextes erleichtern. Da *TMyWindow* als Client in einem Rahmenfenster arbeitet, werden Sie die Klasse *TClientDC* benutzen. *TClientDC* ist eine Gerätekontextklasse, die den Zugriff auf den von einem Fenster besessenen Client-Bereich bietet. Wie alle ObjectWindows-Gerätekontextklassen basiert *TClientDC* auf der Klasse *TDC* und ist in der Header-Datei owl\dc.h definiert.

TClientDC besitzt einen einzigen Konstruktor, der ein HWND als einzigen Parameter erwartet. Da Sie einen Gerätekontext für Ihr *TMyWindow*-Objekt wünschen, benötigen Sie das Handle für dieses Fenster. Hierfür bietet die Basisklasse *TWindow* einen HWND-Konvertierungsoperator. Dieser Operator wird immer dann implizit aufgerufen, wenn Sie das Fensterobjekt an Stellen benutzen, die ein HWND erfordern. Damit sieht der Konstruktor für Ihr *TClientDC*-Objekt in etwa wie folgt aus:

```
TClientDC dc(*this);
```

Beachten Sie, daß der **this**-Zeiger dereferenziert wird. Der HWND-Konvertierungsoperator arbeitet nicht mit Zeigern auf Fensterobjekte.

Drucken in den Gerätekontext

Wenn der Gerätekontext einmal eingerichtet ist, müssen Sie den String ausdrucken. Die Klasse *TDC* bietet mehrere Versionen der Funktion

TextOut. Wie bei der Funktion *MessageBox* in Schritt 2 ähneln die in den Gerätekontextklassen enthaltenen *TextOut*-Funktionen der Windows-API-Funktion *TextOut*. Die erste Version von *TextOut* hat exakt das gleiche Aussehen wie die Windows-API-Version mit der Ausnahme, daß der erste HDC-Parameter weggelassen ist:

```
virtual BOOL TextOut(int x, int y, const char far* str, int count=-1);
```

Der HDC-Parameter wird durch das *TDC*-Objekt gefüllt. Die zweite Version von *TextOut* läßt den HDC-Parameter weg und kombiniert die x- und y-Koordinaten in eine einzelne *TPoint*-Struktur:

```
BOOL TextOut(const TPoint& p, const char far* str, int count=-1);
```

Da die Koordinaten für die Funktion *EvLButtonDown* in einem *TPoint*-Objekt übergeben werden, können Sie die zweite Version von *TextOut* benutzen, um die Koordinaten im Fenster zu drucken. Ihre vervollständigte Funktion *EvLButtonDown* sollte in etwa wie folgt aussehen:

```
void TMyWindow::EvLButtonDown(UINT, TPoint& point)
{
  char s[16];
  TClientDC dc(*this);

  wsprintf(s, "(%d,%d)", point.x, point.y);
  dc.TextOut(point, s, strlen(s));
}
```

Sie müssen die Header-Datei string.h einbinden, um die Funktion *strlen* verwenden zu können.

Leeren des Fensters

Die Basisklasse *TWindow* von *TMyWindow* bietet drei verschiedene Funktionen, um das Fenster ungültig zu machen. Zwei dieser Funktionen mit Namen *InvalidateRect* und *InvalidateRgn* gleichen und funktionieren fast genauso wie ihre entsprechenden Windows-API-Versionen. Sie lassen nur die HWND-Parameter weg. Die dritte Funktion, *Invalidate*, macht den gesamten Client-Bereich des Fensters ungültig. *Invalidate* erwartet ein BOOL als einzigen Parameter, der anzeigt, ob der ungültige Bereich bei seiner Aktualisierung gelöscht werden soll. Standardmäßig ist dieser Parameter TRUE.

Deshalb müssen Sie nur *Invalidate* aufrufen, um den gesamten Client-Bereich von *TMyWindow* zu löschen, und entweder TRUE oder gar nichts bei ihrem ersten Parameter angeben. Um den Bildschirm zu löschen, wenn der Benutzer die rechte Maustaste drückt, müssen Sie diesen Aufruf in der Funktion *EvRButtonDown* stattfinden lassen. Die Funktion würde in etwa wie folgt aussehen:

```
void TMyWindow::EvRButtonDown(UINT, TPoint&)
{
  Invalidate();
}
```

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Gerätekontexte und die *TDC*-Klassen sind in Kapitel 15 beschrieben.
- Fensterklassen sind in Kapitel 8 beschrieben.

Schritt 4: Zeichnen in das Fenster

Sie können den Quelltext für Schritt 4 in der Datei STEP04.CPP im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

In diesem Schritt werden Sie die Möglichkeit hinzufügen, eine Linie im Fenster zu zeichnen, indem die linke Maustaste gedrückt und dann gezogen wird. Dazu fügen Sie zwei neue Ereignisse mit Namen WM_MOUSEMOVE und WM_LBUTTONUP der Antworttabelle von *TMyWindow* hinzu. Ebenso benötigen Sie zwei neue Funktionen für die Behandlung dieser Ereignisse. Sie werden ferner ein *TClientDC* * der Klasse hinzufügen.

Hinzufügen neuer Ereignisse

Damit der Benutzer auf dem Fenster zeichnen kann, muß die Anwendung eine Reihe von Ereignissen behandeln:

- Um das Zeichnen der Linie zu starten, müssen Sie darauf achten, ob der Benutzer die linke Maustaste drückt. Für diese Aufgabe ist durch die Behandlung des WM_LBUTTONDOWN-Ereignisses bereits gesorgt.
- Wenn der Benutzer die linke Taste gedrückt hat, müssen Sie die Mausbewegungen beobachten. An dieser Stelle zeichnen Sie die Linie. Um herauszufinden, wann der Benutzer die Maus bewegt, fangen Sie das Ereignis WM_MOUSEMOVE auf.
- Dann müssen Sie wissen, wann der Benutzer das Zeichnen der Linie beendet hat. Der Benutzer hat das Zeichnen beendet, wenn die linke Maustaste losgelassen wird. Sie können dies mitverfolgen, indem Sie das Ereignis WM_LBUTTONUP auffangen.

Sie müssen zwei Makros in der Antworttabelle der Fensterklasse hinzufügen, EV_WM_MOUSEMOVE und EV_WM_LBUTTONUP. Die neue Antworttabelle sollte in etwa wie folgt aussehen:

```
DEFINE_RESPONSE_TABLE1(TMyWindow, TWindow)
  EV_WM_LBUTTONDOWN,
  EV_WM_RBUTTONDOWN,
  EV_WM_MOUSEMOVE,
  EV_WM_LBUTTONUP,
END_RESPONSE_TABLE;
```

Sie müssen ebenso die Funktionen *EvLButtonUp* und *EvMouseMove* in der Klasse *TMyWindow* hinzufügen.

Hinzufügen eines TClientDC-Zeigers

Das in Schritt 3 verwendete Schema zum Zeichnen einer Linie ist nicht gerade sehr robust:

- In Schritt 3 erzeugten Sie ein *TClientDC*-Objekt in der Funktion *EvLButtonDown*, das automatisch entfernt wurde, wenn die Funktion zurückkehrte. Jetzt benötigen Sie jedoch einen gültigen Gerätekontext über drei verschiedene Funktionen hinweg: *EvLButtonDown*, *EvMouseMove* und *EvLButtonUp*.
- Sie können das Ereignis WM_MOUSEMOVE auffangen und vom aktuellen Punkt bis zu dem Punkt zeichnen, der an die Behandlungsfunktion *EvMouseMove* übergeben wurde. Aber WM_MOUSEMOV-Ereignisse werden jedesmal dann gesendet, wenn die Maus bewegt wurde. Sie wollen jedoch nur dann eine Linie zeichnen, wenn die Maus mit gedrückter linken Taste bewegt wurde.

Sie können beide Probleme sehr einfach lösen, indem Sie ein neues Datenelement unter **protected** für *TMyWindow* hinzufügen. Dieses Datenelement ist ein *TDC* * und trägt den Namen *DragDC*. Die Lösung funktioniert nun wie folgt:

- Wenn die linke Maustaste gedrückt wird, wird die Funktion *EvLButtonDown* aufgerufen. Diese Funktion erzeugt ein neues *TClientDC* und weist es *DragDC* zu. Sie setzt dann den aktuellen Punkt in *DragDC* auf den Punkt, an dem mit der Maus geklickt wurde. Der Code für diese Funktion sollte in etwa wie folgt aussehen:

```
void TMyWindow::EvLButtonDown(UINT, TPoint& point)
{
  Invalidate();
  if (!DragDC) {
    SetCapture();
    DragDC = new TClientDC(*this);
    DragDC->MoveTo(point);
  }
}
```

- Wenn die linke Maustaste losgelassen wird, wird die Funktion *EvLButtonUp* aufgerufen. Wenn *DragDC* gültig ist (d.h., wenn es einen gültigen Gerätekontext repräsentiert) löscht *EvLButtonUp* das Element, indem die Funktion es auf Null setzt. Der Code dazu sollte in etwa wie folgt aussehen:

```
void TMyWindow::EvLButtonUp(UINT, TPoint&)
{
  if (DragDC) {
    ReleaseCapture();
    delete DragDC;
    DragDC = 0;
  }
}
```

- Wenn die Maus bewegt wird, wird die Funktion *EvMouseMove* aufgerufen. Diese Funktion überprüft, ob die linke Maustaste gedrückt ist, indem sie *DragDC* abfragt. Wenn *DragDC* Null ist, wurde die Maustaste entweder gar nicht gedrückt oder gedrückt und losgelassen. In beiden Fällen führt der Benutzer keine Zeichnung aus und die Funktion kehrt zurück. Wenn *DragDC* gültig ist (was bedeutet, daß die linke Maustaste aktuell gedrückt ist), zeichnet die Funktion eine Linie vom aktuellen Punkt zum neuen Punkt mit Hilfe der Funktion *TWindow::LineTo*.

```
void TMyWindow::EvMouseMove(UINT, TPoint& point)
{
  if (DragDC)
    DragDC->LineTo(point);
}
```

Initialisieren von DragDC

Sie müssen sicherstellen, daß *DragDC* auf Null gesetzt ist, wenn das *TMyWindow*-Objekt konstruiert wird:

```
TMyWindow::TMyWindow(TWindow *parent)
{
  Init(parent, 0, 0);
  DragDC = 0;
}
```

Aufräumen nach DragDC

Da *DragDC* ein Zeiger auf ein *TClientDC*-Objekt und nicht ein wirkliches *TClientDC*-Objekt ist, wird es nicht automatisch entfernt, wenn das *TMyWindow*-Objekt entfernt wird. Sie müssen einen Destruktor für *TMyWindow* hinzufügen, der geeignet aufräumt. Das einzig dazu Notwendige ist ein Aufruf von **delete** für *DragDC*. *TMyWindow* sollte nun in etwa wie folgt aussehen:

```
class TMyWindow : public TWindow
{
  public:
    TMyWindow(TWindow *parent = 0);
   ~TMyWindow() {delete DragDC;}

  protected:
    TDC *DragDC;

    // Überschreibe Elementfunktion von TWindow
    BOOL CanClose();

    // Botschaftsantwortfunktionen
    void EvLButtonDown(UINT, TPoint&);
    void EvRButtonDown(UINT, TPoint&);
    void EvMouseMove(UINT, TPoint&);
    void EvLButtonUp(UINT, TPoint&);
```

```
    DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

Beachten Sie, daß der Name des Hauptfensters von »Sample ObjectWindows Program« in »Drawing Pad« geändert wurde, da die Anwendung nun etwas sinnvoller wurde:

```
SetMainWindow(new TFrameWindow(0, "Drawing Pad", new TMyWindow));
```

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Gerätekontexte und die *TDC*-Klassen sind in Kapitel 15 beschrieben.
- Ereignisbehandlung ist in Kapitel 5 beschrieben.
- Vordefinierte Antworttabellenmakros und deren korrespondierende Ereignisbehandlungsfunktionen sind im ObjectWindows *Referenzhandbuch* aufgelistet.

Schritt 5: Ändern der Liniendicke

Sie können den Quelltext für Schritt 5 in den Dateien STEP05.CPP und STEP05.RC im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

In diesem Schritt soll die Fähigkeit zum Zeichnen in der Anwendung etwas robuster werden. Dieser Schritt fügt die Möglichkeit hinzu, die Dicke der Linie zu ändern. Zu diesem Zweck können Sie für die Klasse *TMyWindow* ein Zeichenobjekt *TPe* ***** und ein **int** für die Stiftdicke hinzufügen.

Hinzufügen eines Stiftes

Fügen Sie den Stift zur Fensterklasse in Form von zwei **protected**-Elementen *Pen* (ein *TPen* *****)und *PenSize* (ein **int**) hinzu. Die wichtigsten Änderungen, die aus dem Erweitern der Fensterklasse um diesen Stift resultieren, sind in den Funktionen *EvLButtonDown* und *EvRButtonDown* implementiert.

Initialisieren des Stiftes

Das *Pen*-Objekt und *PenSize* müssen zuerst erzeugt und initialisiert werden, bevor der Benutzer mit dem Stift zeichnen kann. Der beste Ort dafür liegt im Konstruktor:

```
TMyWindow::TMyWindow(TWindow *parent)
{
  Init(parent, 0, 0);
  DragDC = 0;
  PenSize = 1;
  Pen = new TPen(TColor::Black, PenSize);
}
```

Das *TColor::Black*-Objekt im Konstruktor von *TPen* ist ein **enum**, das in der Header-Datei owl\color.h definiert ist. Dadurch wird der Stift schwarz. Sie werden mehr über diesen Parameter des Konstruktors von *TPen* später in Schritt 9 lernen.

Auswählen des Stiftes für DragDC

Um das neue Stiftobjekt für das Ziehen einer Linie zu benutzen, muß der Stift für den Gerätekontext ausgewählt werden. Die Gerätekontextklassen besitzen eine Funktion mit Namen *SelectObject*. Diese Funktion ähnelt der Windows-API-Funktion *SelectObject*, mit der Ausnahme, daß die ObjectWindows-Version kein Handle auf den Gerätekontext erfordert.

Mit *SelectObject* können Sie viele verschiedene Objekte für einen Gerätekontext auswählen, einschließlich Pinsel, Schriften, Paletten und Stiften. Sie müssen *SelectObject* aufrufen, bevor Sie mit dem Zeichnen beginnen. Fügen Sie den Aufruf in der Funktion *EvLButtonDown* direkt hinter dem Erzeugen des Gerätekontextes an:

```
void TMyWindow::EvLButtonDown(UINT, TPoint& point)
{
  Invalidate();

  if (!DragDC) {
    SetCapture();
    DragDC = new TClientDC(*this);
    DragDC->SelectObject(*Pen);
    DragDC->MoveTo(point);
  }
}
```

Beachten Sie, daß *Pen* im Aufruf von *SelectObject* dereferenziert wird, da die Funktion *SelectObject* ein *TPen* **&** als Parameter erwartet und *Pen* ein *TPen* ***** ist. Das Dereferenzieren des Zeigers paßt *Pen* den Typanforderungen von *SelectObject* entsprechend an.

Ändern der Stiftgröße

Die Möglichkeit zum Ändern der Stiftgröße in der Anwendung ist wenig sinnvoll, solange der Benutzer nicht die Zugriffsmöglichkeit besitzt. Um diesen Zugriff bereitzustellen, können Sie die Bedeutung für das Drücken der rechten Maustaste ändern. Anstatt den Bildschirm zu löschen, zeigt sie nun an, daß der Benutzer die Breite des Zeichenstiftes ändern will. Deshalb schlägt sich der Prozeß der Änderung der Stiftgröße in der Funktion *EvRButtonDown* nieder.

Wenn der Benutzer einmal durch Drücken der rechten Maustaste angezeigt hat, daß er die Stiftbreite ändern will, müssen Sie eine Möglichkeit finden, daß der Benutzer die neue Stiftbreite eingeben kann. Dazu können Sie einen *TInputDialog* anzeigen, in den der Benutzer die Stiftgröße eingeben kann.

Konstruieren eines Eingabedialogfensters

Der Konstruktor *TInputDialog* sieht wie folgt aus:

```
TInputDialog(TWindow* parent,
             const char far* title,
             const char far* prompt,
             char far* buffer,
             int bufferSize,
             TModule* module = 0);
```

Dabei gilt folgendes:

- *parent* ist ein Zeiger auf das dem Dialogfenster übergeordnete Fenster. In diesem Fall ist das übergeordnete Fenster das *TMyWindow*-Fenster. Sie können es übergeben, indem Sie einfach den **this**-Zeiger verwenden.
- *title* und *prompt* sind Meldungen, die dem Benutzer angezeigt werden, wenn das Dialogfenster geöffnet wird. In diesem Fall ist *title* (sie wird in die Titelleiste des Dialogfensters plaziert) »Line Thickness«, und *prompt* (sie wird rechts oberhalb des Eingabefeldes plaziert) »Input a new thickness:«.
- *buffer* ist ein String. Dieser String kann vor der Verwendung von *TInputDialog* initialisiert werden. Wenn *buffer* einen gültigen String enthält, wird er in *TInputDialog* als vorgegebene Antwort angezeigt. Initialisieren Sie in diesem Fall *buffer* mit der aktuellen Stiftgröße, die in *PenSize* vorliegt.
- *bufferSize* ist die Größe von *buffer* in Bytes. Am einfachsten können Sie entweder ein **#define**, um Speicher für *buffer* zu reservieren, oder **sizeof**(*buffer*) verwenden.
- *module* wird in diesem Beispiel nicht benutzt.

Um *TInputDialog* zu verwenden, müssen Sie sicherstellen, daß dessen Ressourcen und Ressourcen-Bezeichner in Ihren Quelltextdateien und Ressourcen-Script-Dateien mit eingebunden sind. Diese sind in der Datei include\owl\inputdia.rc enthalten. Sie sollten owl\inputdia.rc in Ihren Ressourcen-Script-Dateien und Ihren C++ Quelltextdateien per **include** einbinden.

Ausführen eines Eingabedialogfensters

Wenn Sie ein *TInputDialog*-Objekt konstruiert haben, können Sie entweder die Funktion *TDialog::Execute* aufrufen, um das Dialogfenster modal auszuführen, oder die Funktion *TDialog::Create* aufrufen, um das Dialogfenster nicht-modal auszuführen. Da es nicht erforderlich ist, das Dialogfenster nicht-modal auszuführen, können Sie die Funktion *Execute* verwenden.

Die Funktion *Execute* kann für *TInputDialog* zwei wichtige Werte zurückgeben, IDOK und IDCANCEL. Der zurückgegebene Wert hängt davon ab, welcher Schalter vom Benutzer gedrückt wurde. Wenn der Benutzer

den »OK«-Schalter drückt, gibt *Execute* IDOK zurück. Wenn der Benutzer den »Abbruch«-Schalter drückt, gibt *Execute* IDCANCEL zurück. Deshalb müssen Sie bei der Ausführung des Eingabedialogfensters überprüfen, ob der Rückgabewert IDOK ist, bevor Sie die Stiftgröße ändern. Wenn der Rückgabewert nicht IDOK ist, dann lassen Sie die Stiftgröße wie sie war.

Wenn der Aufruf von *Execute* IDOK zurückgibt, steht der neue Wert für *PenSize* in dem String, der für den Puffer des Dialoges übergeben wurde. Bevor der Wert als Stiftgröße benutzt werden kann, muß er in ein **int** konvertiert werden. Dann sollten Sie sicherstellen, daß der vom Puffer erhaltene Wert eine gültige Stiftbreite angibt. Wenn die Benutzereingabe korrekt ist, können Sie schließlich die Stiftgröße ändern. *TMyWindow* besitzt jetzt eine Funktion mit Namen *SetPenSize*, die Sie für das Ändern der Stiftgröße verwenden können. Der Grund dafür, auf diese Weise vorzugehen und nicht den Stift direkt zu ändern, wird im nächsten Abschnitt erklärt.

Die Funktion *EvRButtonDown* sollte nun in etwa wie folgt aussehen:

```
void TMyWindow::EvRButtonDown(UINT, TPoint&)
{
  char inputText[6];

  wsprintf(inputText, "%d", PenSize);
  if ((TInputDialog(this, "Line Thickness",
                        "Input a new thickness:",
                        inputText,
                        sizeof(inputText))).Execute() == IDOK) {
    int  newPenSize = atoi(inputText);

    if (newPenSize < 0)
      newPenSize = 1;

    SetPenSize(newPenSize);
  }
}
```

Aufrufen von SetPenSize

Um die Stiftgröße zu ändern, verwenden Sie die Funktion *SetPenSize*. Obwohl die Funktion *EvRButtonDown* ein Element von *TMyWindow* ist und als solches vollen Zugriff auf die **protected**-Datenelemente *Pen* und *PenSize* besitzt, ist es besser, eine öffentliche Zugriffsfunktion für die aktuellen Änderungen an den Daten einzurichten. Dies wird später wichtig, wenn der Stift öfter geändert wird.

Für *TMyWindow* haben Sie die **public**-Funktion *SetPenSize*. Die Funktion *SetPenSize* erwartet ein **int** als einzigen Parameter, der die neue Breite für den Stift enthält. Nach dem Öffnen des Eingabedialogfensters, dem Verarbeiten der Eingabe und dem Prüfen der Gültigkeit des Ergebnisses müssen Sie nur noch *SetPenSize* aufrufen.

SetPenSize ist eine sehr einfache Funktion. Um die Größe des Stiftes anzupassen, müssen Sie zuerst das existierende Stiftobjekt löschen. Dann setzen Sie *PenSize* auf die neue Größe. Zuletzt konstruieren Sie ein neues Stiftobjekt mit den neuen Stiftgröße. Die Funktion sollte in etwa wie folgt aussehen:

```
void TMyWindow::SetPenSize(int newSize)
{
  delete Pen;
  PenSize = newSize;
  Pen = new TPen(TColor(0,0,0), PenSize);
}
```

Aufräumen von Pen

Da *Pen* ein Zeiger auf ein *TPen*-Objekt und kein wirkliches *TPen*-Objekt ist, wird es nicht automatische entfernt, wenn das *TMyWindow*-Objekt entfernt wird. Sie müssen *Pen* explizit im Destruktor von *TMyWindow* für ein vollständiges Aufräumen entfernen. Sie müssen allerdings dazu nur **delete** für Pen aufrufen. *TMyWindow* sollte nun in etwa wie folgt aussehen:

```
class TMyWindow : public TWindow
{
  public:
    TMyWindow(TWindow *parent = 0);
   ~TMyWindow() {delete DragDC; delete Pen;}

    void SetPenSize(int newSize);

  protected:
    TDC *DragDC;
    int PenSize;
    TPen *Pen;

    // Überschreibe Elementfunktion von TWindow
    BOOL CanClose();

    // Botschaftsantwortfunktionen
    void EvLButtonDown(UINT, TPoint&);
    void EvRButtonDown(UINT, TPoint&);
    void EvMouseMove(UINT, TPoint&);
    void EvLButtonUp(UINT, TPoint&);

    DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Gerätekontexte und die *TDC*-Klassen sind in Kapitel 15 beschrieben.
- Die Klasse *TPen* ist in Kapitel 15 beschrieben.
- Die Klasse *TInputDialog* und Dialoge im allgemeinen sind in Kapitel 10 beschrieben.

Schritt 6: Zeichnen des Fensters und Hinzufügen von Menüs

Bei der Anwendung aus Schritt 5 gibt es einige Unstimmigkeiten. Das größte Problem besteht darin, daß das darstellende Fenster nicht weiß, wie es sich selbst zeichnen soll. Zeichnen Sie zur Verdeutlichung des Problems eine Linie in das Fenster, machen Sie die Anwendung zu einem Symbol und stellen Sie das Fenster dann wieder her. Die von Ihnen gezeichnete Linie ist dann verschwunden.

Sie können den Quelltext für Schritt 6 in den Dateien STEP06.CPP und STEP06.RC im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

Ein anderes Problem besteht darin, daß der Benutzer auf die Anwendung nur mit der Maus zugreifen kann. Er kann entweder die linke Maustaste drücken, um eine Linie zu ziehen, oder die rechte, um die Stiftgröße zu ändern.

In Schritt 6 werden Sie die Anwendung in die Lage versetzen, sich an den Inhalt des Fensters zu erinnern und ihn neu zu zeichnen. Sie werden ferner einige Menüs hinzufügen, um die Zugriffsmöglichkeiten für den Benutzer zu erweitern.

Neuzeichnen des Fensters

Wenn Sie das Fenster zeichnen wollen, müssen Sie zwei Probleme lösen:

- Es muß eine Möglichkeit geben, die Information darüber abzurufen, was im Fenster angezeigt wurde.
- Es muß eine Möglichkeit geben, das Fenster neuzuzeichnen.

Speichern der Zeichnung

In den früheren Lektionsschritten der Anwendung wurde die Linie im Fenster gezogen, wenn der Benutzer die Maus mit gedrückter linker Maustaste bewegt hat. Dieser Ansatz reicht für das Zeichnen der Linie aus, er speichert jedoch nicht die Punkte der Linie für spätere Zwecke.

Da sich die Linie aus einer Reihe von Punkten im Fenster zusammensetzt, können Sie jeden Punkt in der ObjectWindows-Klasse *TPoint* speichern. Da ferner jede Linie aus mehreren Punkten besteht, benötigen Sie ein Array mit *TPoint*-Objekten zur Speicherung der Linie. Anstatt Speicher für ein Array mit *TPoint*-Objekten zu reservieren, dieses Array zu verwalten und zu aktualisieren, greift die hier vorgestellte Anwendung auf die Containerklasse *TArray* von Borland zurück, um einen Datentyp mit Namen *TPoints* zu definieren. Sie verwendet zudem die Containerklasse *TArrayIterator* von Borland, um einen Iterator mit Namen *TPointsIterator* zu definieren. Die Definitionen dieser beiden Typen sehen wie folgt aus:

```
typedef TArray<TPoint> TPoints;
typedef TArrayIterator<TPoint> TPointsIterator;
```

Die Klasse *TMyWindow* fügt ein *TPoints*-Objekt hinzu, in dem Sie die Punkte der Linie speichern kann. Sie benutzt in Wirklichkeit ein *TPoints* * , ein **protected**-Element mit Namen *Line*. Dieses Element zeigt auf ein *TPoints*-Array, das im Konstruktor erzeugt wird. Der Konstruktor sieht nun in etwa wie folgt aus:

```
TMyWindow::TMyWindow(TWindow *parent)
{
  Init(parent, 0, 0);
  DragDC = 0;
  PenSize = 1;
  Pen = new TPen(TColor::Black, PenSize);
  Line = new TPoints(10, 0, 10);
}
```

TPoints

Die Container-Klassenbibliothek von Borland C++ und die Klassen *TArray* und *TArrayIterator* sind im Detail im Borland C++ *Programmierhandbuch* erklärt. Für die hier verfolgten Zwecke reicht eine einfache Erklärung, wie die Containerklassen *TPoints* und *TPointsIterator* in der Beispielanwendung benutzt werden. Um die Klassen *TArray* und *TArrayIterator* zu verwenden, müssen Sie die Header-Datei classlib\arrays.h einbinden.

Der Konstruktor *TArray* erwartet drei Parameter, die alle vom Typ **int** sind:

- Der erster Parameter repräsentiert die Obergrenze des Array. Das heißt, wie hoch der Array-Zähler gehen kann.
- Der zweite Parameter repräsentiert die Untergrenze des Array. Das heißt, die Zahl, bei welcher der Array-Zähler beginnt. Dieser Parameter ist standardmäßig Null, was mit der C- und C++ Konvention übereinstimmt, daß Arrays bei Element Null starten.
- Der dritte Parameter repräsentiert den Delta-Wert des Array. Dieser Delta-Wert ist die Zahl der Elemente, um die das Array erweitert wird, wenn es zu groß wird, um alle Elemente des Array zu enthalten.

Hier ist die Anweisung, die Speicher für das anfängliche Array mit Punkten im Konstruktor von *TMyWindow* reserviert:

```
Line = new TPoints(10, 0, 10);
```

Das Punkte-Array wird mit Platz für zehn Elemente erzeugt, die bei Null beginnen. Wenn zehn Objekte im Array gespeichert sind, verursacht das Hinzufügen eines weiteren Objektes eine Platzreservierung für zehn neue Elemente im Array. Damit können Sie mit einem kleinen Array beginnen. Es erleichtert jedoch auch eines der Hauptprobleme, die man mit statischen Arrays hat, wenn sie nicht genug Elemente aufnehmen können und man deshalb Platz neu reservieren muß, um das Array zu erweitern.

Wenn Sie ein Array erzeugt haben, müssen Sie es auch manipulieren können. Die Klasse *TArray* (und durch Erweiterung die Klasse *TPoints*) bietet eine Reihe von Funktionen, um Elemente hinzuzufügen, sie zu löschen, das Array zu leeren und ähnliches mehr. Die Beispielanwendung greift nur auf eine kleine Zahl der zur Verfügung stehenden Funktionen zurück. Hier ist eine kurze Beschreibung jeder Funktion:

- Die Funktion *Add* fügt dem Array ein Element hinzu. Sie erwartet einen einzelnen Parameter, eine Referenz auf ein Objekt des Array-Typs. Das Hinzufügen eines *TPoint*-Objektes zu einem *TPoints*-Array würde beispielsweise in etwa wie folgt aussehen:

```
// Konstruiere ein TPoints-Array (ein Array mit TPoint-Objekten)
TPoints Points(10, 0, 10);

// Konstruiere ein TPoint-Objekt
TPoint p(3,4);

// Füge das TPoint-Objekt p dem Array hinzu
Points.Add(p);
```

- Die Funktion *Flush* leert alle Elemente eines Array und setzt die Zahl von Array-Elementen auf die Anfangsgröße zurück. Sie erwartet keine Parameter. Um das Array im oben stehenden Beispiel-Code zu leeren, würde der Funktionsaufruf in etwa wie folgt aussehen:

```
// Leere alle Elemente des Points-Array
Points.Flush();
```

- Die Funktion *GetItemsInContainer* gibt die Gesamtzahl von Elementen im Container zurück. Beachten Sie, daß diese Zahl die Zahl der zum Container aktuell hinzugefügten Objekte und nicht den verfügbaren Platz repräsentiert. Auch wenn zum Beispiel der Container genügend Platz für 30 Objekte haben kann, könnte er nur 23 Objekte enthalten. In diesem Fall würde *GetItemsInContainer* 23 zurückgeben.

TPointsIterator

Iteratoren – in diesem Fall der *TPointsIterator*-Typ – ermöglichen Ihnen das Bewegen durch das Array, wobei sie jeweils auf ein einzelnes Array-Element zugreifen. Ein Iterator-Konstruktor erwartet als einzigen Parameter eine Referenz auf ein *TArray* mit Objekten (Der Typ der Objekte im Array wird durch die Definition des Iterator eingestellt). Wenn der Iterator auf die Verwendung des Elementes *Line* der Klasse *TMyWindow* eingestellt ist, sieht er wie folgt aus:

```
TPointsIterator i(*Line);
```

Beachten Sie, daß *Line* dereferenziert wird, da der Iterator-Konstruktor ein *TPoints* **&** als ersten Parameter erwartet und *Line* ein *TPoints* ***** ist. Das Dereferenzieren des Zeigers paßt *Line* den Typanforderungen des Iterator-Konstruktors entsprechend an.

Wenn Sie einen Iterator erzeugt haben, können Sie ihn für den Zugriff auf jedes Objekt im Array verwenden. Der Zugriff startet beim ersten Element und betrifft immer nur jeweils ein Element. In der Beispielanwendung wird der Iterator nicht besonders intensiv genutzt und Sie können damit nicht viel über die Möglichkeiten eines Iterators daraus lernen. Aber das Beispiel greift auf Eigenschaften von Iteratoren zurück, die einer Erläuterung verlangen:

- Sie können sich durch die Objekte im Array bewegen, indem Sie den Operator **++** auf den Iterator anwenden. Dieser gibt eine Referenz auf das aktuelle Objekte zurück und inkrementiert den Iterator zum nächsten Objekt im Array. Die Reihenfolge, in der der Operator diese beiden Aktionen ausführt, hängt davon ab, ob Sie **++** als Präfix- oder Postfixoperator verwenden. Beim Präfixoperator (z.B. **++**i) wird der Iterator auf das nächste Objekt hin inkrementiert und eine Referenz auf dieses Objekt zurückgegeben. Beim Postfixoperator (z.B. ***i*++**) wird eine Referenz auf das aktuelle Objekt zurückgegeben und dann der Iterator auf das nächste Objekt hin inkrementiert.

Wenn Sie den Iterator nach dem letzten Element des Array inkrementieren, wird der Iterator auf Null gesetzt. Sie können dies für einen Test in einem beliebigen booleschen Ausdruck ausnutzen. Ein Beispiel:

```
TPointsIterator i(*Line);

while(i)
  i++;
```

- Sie können ferner auf das aktuelle Objekt mit der Funktion *Current* zugreifen. Ein Aufruf dieser Funktion gibt eine Referenz auf das aktuelle Objekt zurück. Sie können dann Operationen auf dem Objekt ausführen, als wenn es eine reguläre Instanz des Objektes wäre. Sie können zum Beispiel einen Punkt, auf den durch einen Iterator zugegriffen wurde, mit einem anderen Punkt vergleichen:

```
TPointsIterator i(*Line);

TPoint tmp(5, 6);

if (i.Current() == tmp)
  return TRUE;
else
  return FALSE;
```

Verwenden der Array-Klassen

Wenn ein *Line*-Array im Konstruktor *TMyWindow* erzeugt ist, wird darauf an vier wichtigen Stellen zugegriffen:

- Die Funktion *EvLButtonDown*. Das Array wird zu Beginn der Funktion geleert, bevor der Bildschirm ungültig gemacht wird. Der Startpunkt der Linie wird dann gegen Ende der Funktion in die Linie eingefügt. Die Funktion *EvLButtonDown* sollte in etwa wie folgt aussehen:

```
void TMyWindow::EvLButtonDown(UINT, TPoint& point)
{
  Line->Flush();
  Invalidate();

  if (!DragDC) {
    SetCapture();
    DragDC = new TClientDC(*this);
    DragDC->SelectObject(*Pen);
    DragDC->MoveTo(point);
    Line->Add(point);
  }
}
```

- Die Funktion *EvMouseMove*. Jeder Punkt in der Linie wird dem Array entsprechend der Bewegungen des Benutzers im Fenster hinzugefügt. Die Funktion sollte in etwa wie folgt aussehen:

```
void TMyWindow::EvMouseMove(UINT, TPoint& point)
{
  if (DragDC) {
    DragDC->LineTo(point);
    Line->Add(point);
  }
}
```

- Die Funktion *Paint*. Diese Funktion wird im nächsten Abschnitt beschrieben.
- Die Funktion *CmFileNew*.

Die Funktion Paint

Wenn Sie in C-Windows-Standardprogrammen ein Fenster manuell neu zeichnen müssen, fangen Sie die WM_PAINT-Botschaften auf und führen alles durch, was für das Neuzeichnen des Bildschirms erforderlich ist. Dies könnte zu der Vermutung führen, daß der geeignete Weg für das Neuzeichnen des Fensters in der Klasse *TMyWindow* darin besteht, das Makro EV_WM_PAINT in der Antworttabelle der Klasse hinzuzufügen und eine Funktion mit Namen *EvPaint* einzurichten.

Sie können dies auf diese Weise auch lösen. Ein besserer Weg stellt allerdings das Überschreiben der *TWindow*-Funktion *Paint* dar. Die Basisklasse *TWindow* von *TMyWindow* erledigt in ihrer Funktion *EvPaint* einen großen Teil. Sie richtet Aufrufe von *BeginPaint* und *EndPaint* ein, sie erzeugt einen Gerätekontext für das Fenster usw.

Paint ist ein **virtual**-Element der Klasse *TWindow*. Die Funktion *EvPaint* von *TWindow* ruft sie in der Mitte ihrer Bearbeitung auf. Die Standardfunktion *Paint* hat keinen Effekt. Sie können sie benutzen, um eine spezielle Bearbeitung bereitzustellen, die für das Zeichnen einer Linie ausgehend vom *TPoints*-Array erforderlich ist.

Hier ist der Kopf der Funktion *Paint*. Dieser wird in der Klasse *TMyWindow* hinzugefügt:

```
void Paint(TDC&, BOOL, TRect&);
```

Dabei gilt:

- Der erste Parameter ist der Gerätekontext, der durch die aufrufende Funktion eingerichtet wird. Dies ist der Gerätekontext, den Sie für Ihre Arbeit verwenden sollten.
- Wenn der zweite Parameter TRUE ist, sollten Sie den Gerätekontext löschen, bevor Sie das Fenster zeichnen. Wenn er FALSE ist, sollen Sie über den Inhalt des Fensters zeichnen.
- Der dritte Parameter zeigt den ungültigen Bereich des Gerätekontextes an, der neugezeichnet werden muß.

Im aktuellen Fall wollen Sie das Fenster immer leeren. Sie können ferner davon ausgehen, daß der gesamte Bereich der Zeichnung neu gezeichnet werden muß. Die Funktion *Paint* implementiert den folgenden grundlegenden Algorithmus:

- Erzeuge einen Iterator, um die Punkte der Linie zu durchlaufen.
- Wähle den Stift für den Gerätekontext aus, der an die Funktion *Paint* übergeben wird.
- Wenn es sich um den ersten Punkt im Array handelt, dann setze den aktuellen Punkt auf die Koordinaten, die im aktuellen Array-Element enthalten sind.
- Solange noch Punkte im Array ausstehen, zeichne Linien vom aktuellen Punkt bis zum Punkt, der im aktuellen Array-Element enthalten ist.

Die Funktion *TMyWindow::Paint* sieht nun in etwa wie folgt aus:

```
void TMyWindow::Paint(TDC& dc, BOOL, TRect&)
{
  BOOL first = TRUE;
  TPointsIterator i(*Line);

  dc.SelectObject(*Pen);

  while (i) {
    TPoint p = i++;

    if (!first)
      dc.LineTo(p);
    else {
      dc.MoveTo(p);
      first = FALSE;
    }
  }
}
```

Menükommandos

Um eine Menüauswahl und ihre korrespondierende Ereignisbehandlung für Ihre Anwendung hinzuzufügen, müssen Sie eine Reihe von Schritten ausführen:

- Definieren Sie einen Ereignisbezeichner für die Menüauswahl. Nach Konvention besteht dieser Bezeichner nur aus Großbuchstaben und beginnt mit »CM_«. So lautet zum Beispiel der Bezeichner für die Menüauswahl *Datei / Öffnen* CM_FILEOPEN.
- Fügen Sie die passende Menü-Ressource in ihrer Ressourcendatei hinzu.
- Fügen Sie eine Ereignisbehandlungsfunktion für die Menüauswahl in Ihrer Klasse hinzu. Die Konvention in ObjectWindows 2.5 für den Funktionsnamen sieht eine Übereinstimmung mit dem Ereignisbezeichner vor, allerdings mit der Ausnahme, daß der Unterstrich weggelassen wird und daß Wortteile mit Großbuchstaben beginnen und der Rest klein geschrieben wird. So trägt zum Beispiel die Funktion, die das Ereignis CM_FILEOPEN behandelt, den Namen *CmFileOpen*.
- Fügen Sie ein Makro EV_COMMAND in der Antworttabelle Ihrer Klasse hinzu, welches den Ereignisbezeichner mit der Ereignisbehandlungsfunktion assoziiert. Dieses Makro erwartet zwei Parameter. Der erste ist der Ereignisbezeichner und der zweite ist der Name der Ereignisbehandlungsfunktion. So sieht zum Beispiel der Antworttabelleneintrag für die Menüauswahl *Datei / Öffnen* wie folgt aus:

```
EV_COMMAND(CM_FILEOPEN, CmFileOpen);
```

- Das Makro EV_COMMAND erfordert vom Kopf der Ereignisbehandlungsfunktion, daß sie keine Parameter erwartet und **void** zurückgibt. Damit sieht der Kopf der Ereignisbehandlungsfunktion für die Menüauswahl *Datei / Öffnen* wie folgt aus:

```
void CmFileOpen();
```

Hinzufügen der Ereignisbezeichner

Sie müssen Bezeichner für jede dieser Menüauswahlen hinzufügen. Hier ist die Definition des Ereignisbezeichners:

```
#define CM_FILENEW    201
#define CM_FILEOPEN   202
#define CM_FILESAVE   203
#define CM_FILESAVEAS 204
#define CM_ABOUT      205
```

Diese Bezeichner sind in der Datei STEP06.RC enthalten. Nach ObjectWindows-Stil liegen die Definitionen von Bezeichnern in der Ressourcen-Script-Datei und nicht in einer Header-Datei. Dies reduziert die Zahl der für ein Projekt erforderlichen Quelltextdateien und erleichtert die Verwaltung der Konsistenz der Bezeichnerwerte zwischen den Ressourcen und dem Quelltext der Anwendung.

Die eigentlichen Ressourcen-Definitionen in der Ressourcendatei liegen in einem Block, der in einem **#ifndef/#endif**-Block geklammert ist. In etwa so:

```
#ifdef RC_INVOKED
  // Ressource-Definitionen hier.
  :
#endif
```

RC_INVOKED wird durch alle Ressourcen-Compiler definiert, jedoch nicht durch C++ Compiler. Die Ressourcen-Information ist während der C++ Compilierung niemals sichtbar. Bezeichnerdefinitionen sollten außerhalb dieses **#ifndef/#endif**-Blockes und normalerweise zu Beginn der Datei plaziert werden.

Hinzufügen von Menü-Ressourcen

Für den hier verfolgten Zweck sollen fünf Menüauswahlmöglichkeiten für die Anwendung hinzugefügt werden:

- Datei Neu
- Datei Öffnen
- Datei Speichern
- Datei Speichern unter
- Info

Jede dieser Menüauswahlen muß mit dem korrekten Ereignisbezeichner assoziiert werden. Das heißt, die Menüauswahl *Datei / Öffnen* sollte das Ereignis CM_FILEOPEN senden.

Die Menü-Ressource wird mit der Anwendung in der Funktion *InitMainWindow* verbunden. Sie müssen die Funktion *AssignMenu* des Hauptfensters aufrufen. Um das Hauptfenster zu erhalten, können Sie die Funktion *GetMainWindow* aufrufen. Die Funktion *InitMainWindow* sollte wie folgt aussehen:

```
void InitMainWindow()
{
  SetMainWindow(new TFrameWindow(0, "Drawing Pad", new TMyWindow));
  GetMainWindow()->AssignMenu("COMMANDS");
}
```

Hinzufügen von Antworttabelleneinträgen

Jeder Ereignisbezeichner muß mit seiner korrespondierenden Behandlungsfunktion assoziiert werden. Dazu fügen Sie die folgenden Zeilen in der Antworttabelle hinzu:

```
EV_COMMAND(CM_FILENEW, CmFileNew),
EV_COMMAND(CM_FILEOPEN, CmFileOpen),
EV_COMMAND(CM_FILESAVE, CmFileSave),
EV_COMMAND(CM_FILESAVEAS, CmFileSaveAs),
EV_COMMAND(CM_ABOUT, CmAbout),
```

Hinzufügen von Ereignisbehandlungen

Nun müssen Sie eine Funktion hinzufügen, um jedes Ereignis, das Sie gerade der Antworttabelle hinzugefügt haben, zu behandeln. Da diese Funktionen eventuell sehr groß werden können, sollten Sie sie innerhalb der Klassendeklaration deklarieren und sie außerhalb der Klassendeklaration definieren.

Die Deklarationen dieser Funktionen sollten in etwa wie folgt aussehen:

```
void CmFileNew();
void CmFileOpen();
void CmFileSave();
void CmFileSaveAs();
void CmAbout();
```

Implementieren der Ereignisbehandlungen

Der letzte Schritt beim Implementieren der Ereignisbehandlungen besteht in der Definition der Funktionen. Für den hier verfolgten Zweck sollten Sie die Implementierung der Funktionen auf ein Minimum beschränken. Die meisten von ihnen können gerade ein Meldungsfenster anzeigen, die mitteilen, daß die Funktion noch nicht implementiert ist. Die auf diese Art eingerichteten Funktionen sind *CmFileOpen*, *CmFileSave*, *CmFileSaveAs* und *CmAbout*. Hier ist nun das Aussehen dieser Funktionen:

```
void TMyWindow::CmFileOpen()
{
  MessageBox("Feature not implemented", "File Open", MB_OK);
}
```

Die einzige in diesem Schritt implementierte Funktion ist *CmFileNew*, da sie sehr leicht einzurichten ist. Ihre einzige Aufgabe besteht in dem Löschen des Array mit Punkten und dem Löschen des Fensters. Die Funktion *CmFileNew* sieht wie folgt aus:

```
void TMyWindow::CmFileNew()
{
  Line->Flush();
  Invalidate();
}
```

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Fensterklassen sind in Kapitel 8 beschrieben.
- Die Containerklassenbibliothek von Borland C++ und die Klassen *TArray* und *TArrayIterator* sind in Kapitel 9 im Borland C++ Programmierhandbuch erklärt.
- Menüs und Menüobjekte sind in Kapitel 9 erklärt.
- Ereignisbehandlung ist in Kapitel 5 beschrieben.

Schritt 7: Verwenden allgemeiner Dialogfenster

In diesem Schritt werden Sie die Ereignisbehandlungsfunktionen implementieren, die Sie in Schritt 6 hinzugefügt haben. Die Funktion *CmFileOpen*, die Funktion *CmFileSave* und die Funktion *CmFileSaveAs* verwenden die ObjectWindows-Klassen *TFileOpenDialog* und *TFileSaveDialog*. Diese Klassen kapseln die allgemeinen Dialogfenster von Windows zum Öffnen und Speichern, um den Benutzer zur Eingabe von Dateinamen aufzufordern.

Sie können den Quelltext für Schritt 7 in den Dateien STEP07.CPP und STEP07.RC im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

Sie werden die Funktion *CanClose* dahingehend ändern, daß Sie vor dem Entfernen der Zeichnung überprüfen, ob sich die Zeichnung im Fenster geändert hat. Ist dies der Fall, erhält der Benutzer die Möglichkeit, die Datei entweder zu speichern, ohne eine Speicherung der Datei fortzufahren oder die gesamte Operation zum Schließen abzubrechen.

Ferner werden Sie zur Implementierung der Funktionen *CmFileOpen*, *CmFileSave* und *CmFileSaveAs* zwei weitere Funktionen als **protected**-Elemente mit Namen *OpenFile* und *SaveFile* für die Fensterklasse hinzufügen. Diese Funktionen werden später in diesem Schritt beschrieben.

Änderungen an TMyWindow

Fügen Sie für die Implementierung der Menükommandos einige neue Datenelemente für die Klasse *TMyWindow* hinzu: *FileData*, *IsDirty* und *IsNewFile*.

FileData

Das Element *FileData* ist ein Zeiger auf ein *TOpenSaveDialog::TData*-Objekt. Die Klasse *TOpenSaveDialog* ist die direkte Basisklasse sowohl von der Klasse *TFileOpenDialog* als auch von der Klasse *TFileSaveDialog*. Beide Klassen verwenden die Klasse *TOpenSaveDialog::TData*, für die Aufnahme von Informationen bezüglich der aktuellen Datei oder Dateioperation, wie zum Beispiel dem Dateinamen, dem für die Suche voreingestellten Verzeichnis, den Dateinamensfiltern usw.

FileData wird mit **new** im Konstruktor *TMyWindow* mit einem neuen *TOpenSaveDialog::TData*-Objekt initialisiert. Da *FileData* ein Zeiger auf ein Objekt ist, muß der Destruktor von *TMyWindow* eine **delete**-Anweisung erhalten, um sicherzustellen, daß das Objekt aus dem Speicher entfernt wird, wenn die Anwendung terminiert.

IsDirty

Das Flag *IsDirty* zeigt an, ob die aktuelle Zeichnung »dirty« ist; das heißt, ob die Zeichnung seit der letzten Änderung durch den Benutzer gespeichert wurde. Wenn die Zeichnung nicht geändert wurde oder wenn der Benutzer noch nichts im leeren Fenster gezeichnet hat, ist *IsDirty* auf FALSE gesetzt, anderenfalls auf TRUE. *IsDirty* wird im Konstruktor *TMyWindow* auf FALSE gesetzt, da die Zeichnung noch nicht geändert wurde.

Außerhalb des Konstruktors wird das Flag *IsDirty* in einer Reihe von Funktionen gesetzt:

- In der Funktion *EvLButtonDown* wird *IsDirty* auf FALSE, um die an der Zeichnung vorgenommene Änderung zu reflektieren.
- In der Funktion *CmFileNew* wird *IsDirty* auf FALSE gesetzt, wenn das Fenster geleert wird.
- In den Funktionen *OpenFile* und *SaveFile* wird *IsDirty* auf FALSE gesetzt, um anzuzeigen, daß die Zeichnung seit dem letzten Speichern oder Laden nicht verändert wurde.

IsNewFile

Das Flag *IsNewFile* zeigt an, ob die Datei einen Namen besitzt. Eine Datei besitzt einen Namen, wenn sie von einer existierenden Datei aus geladen wurde oder wenn sie mit einem bestimmten Dateinamen auf Platte oder Diskette gesichert wurde. Wenn die Datei einen Namen besitzt (das heißt, wenn sie vorher gespeichert wurde oder von einer existierenden Datei geladen wurde), steht das Flag *IsNewFile* auf FALSE. *IsNewFile* ist im Konstruktor *TMyWindow* auf TRUE gesetzt, da die Zeichnung noch nicht mit einem Namen gespeichert wurde.

Außerhalb des Konstruktors wird das Flag *IsNewFile* in einer Reihe von Funktionen gesetzt:

- In der Funktion *CmFileNew* wird *IsNewFile* auf TRUE gesetzt, wenn das Fenster geleert wird.
- In den Funktionen *OpenFile* und *SaveFile* wird *IsNewFile* auf FALSE gesetzt, um anzuzeigen, daß die Zeichnung auf Platte oder Diskette gespeichert wurde.

Verbessern von CanClose

Die Funktion *CanClose*, die Sie seit Schritt 2 dieser Einführung bereits nutzen, besitzt eine Reihe von Fehlern. Erstens fragt sie bei jedem Aufruf den Benutzer, ob die Zeichnung gespeichert werden soll. Dies ist nicht notwendig, wenn die Zeichnung seit dem letzten Speichern oder Laden nicht mehr geändert wurde oder wenn das Fenster geleert wurde. Zweitens ist eine einfache Ja/Nein-Antwort für diese Frage nicht ausreichend. Wenn der Benutzer zum Beispiel das Fenster nicht schließen will, wäre die gewünschte Antwort, die ganze Operation abzubrechen.

Eine Überprüfung des Flag *IsDirty* teilt der Funktion *CanClose* mit, ob es überhaupt notwendig ist, den Benutzer zu einer Billigung der Operation zum Schließen aufzufordern. Wenn die Zeichnung nicht geändert wurde, besteht dazu keine Veranlassung. Der Benutzer kann die Datei einfach neu laden.

Wenn die Datei verändert wurde, zeigt die Funktion *CanClose* ein Meldungsfenster an. Die Verwendung des Flag MB_YESNOCANCEL im Auf-

ruf des Meldungsfensters gibt dem Benutzer eine Auswahl zwischen drei an Stelle von zwei Möglichkeiten:

- Die Auswahl von »Abbruch« bedeutet, daß der Benutzer die ganze Operation zum Schließen abbrechen will. In diesem Fall gibt *MessageBox* IDCANCEL und die Funktion *CanClose* FALSE zurück. Damit wird der aufrufenden Funktion signalisiert, daß ein Fortfahren der Operation *nicht* gewünscht ist.
- Die Auswahl von »Ja« bedeutet, daß der Benutzer die Datei vor dem Fortfahren speichern will. *MessageBox* gibt IDYES zurück und die Funktion *CanClose* ruft die Funktion *CmFileSave* auf (*CmFileSave* wird später in diesem Abschnitt erklärt). Nach dem Aufruf von *CmFileSave* gibt *CanClose* TRUE zurück, womit der aufrufenden Funktion signalisiert wird, daß ein Fortfahren erlaubt ist.
- Die Auswahl von »Nein« bedeutet, daß der Benutzer die Datei vor dem Fortfahren nicht speichern will. In diesem Fall führt *CanClose* keine weiteren Aktionen durch und gibt TRUE zurück.

Der Code für die neue Funktion *CanClose* sieht in etwa wie folgt aus:

```
BOOL TMyWindow::CanClose()
{
  if (IsDirty)
    switch(MessageBox("Do you want to save?", "Drawing has changed",
                      MB_YESNOCANCEL | MB_ICONQUESTION)) {
      case IDCANCEL:
        // Eine Auswahl von Abbruch bedeutet ein Abbrechen des Schließens
        // -  gib FALSE zurück.
        return FALSE;

      case IDYES:
        // Eine Auswahl von Ja bedeutet die Zeichnung zu speichern.
        CmFileSave();
    }
  return TRUE;
}
```

Beachten Sie, daß die Funktion *CmFileNew* in diesem Schritt geändert wurde, um die neue Funktion *CanClose* entsprechend nutzen zu können.

Die Funktion CmFileSave

Die Funktion *CmFileSave* ist relativ einfach. Sie prüft über *IsNewFile*, ob die Zeichnung neu ist. Wenn *IsNewFile* TRUE ist, ruft *CmFileSave* die Funktion *CmFileSaveAs* auf, welche den Benutzer zur Eingabe einer Datei auffordert, in welche die Zeichnung gespeichert werden soll. Anderenfalls ruft sie *SaveFile* auf, welche die Speicherung der Zeichnung durchführt.

Die Funktion *CmFileSave* sollte in etwa wie folgt aussehen:

```
void TMyWindow::CmFileSave()
{
```

```
  if (IsNewFile)
    CmFileSaveAs();
  else
    SaveFile();
}
```

Die Funktion CmFileOpen

Die Funktion *CmFileOpen* ist ebenfalls sehr einfach. Sie überprüft zuerst *CanClose*, um sicherzustellen, ob ein Schließen der aktuellen Zeichnung und ein Öffnen einer neuen Datei erlaubt ist. Wenn die Funktion *CanClose* FALSE zurückgibt, bricht *CmFileOpen* ab.

Nachdem sichergestellt ist, daß ein Fortfahren erlaubt ist, erzeugt *CmFileOpen* ein *TFileOpenDialog*-Objekt. Der Konstruktor *TFileOpenDialog* kann bis zu fünf Parameter erwarten. Für diese Anwendung benötigen Sie allerdings nur einen mit zwei Parametern. Die letzten drei Parameter besitzen alle Vorgabewerte. Die zwei benötigten Parameter sind ein Zeiger auf das übergeordnete Fenster und eine Referenz auf ein *TOpenSaveDialog::TData*-Objekt. In diesem Fall ist der Zeiger auf das übergeordnete Fenster der **this**-Zeiger. Das *TOpenSaveDialog::TData*-Objekt wird durch *FileData* bereitgestellt.

Wenn das Dialogfensterobjekt konstruiert ist, wird es durch einen Aufruf der Funktion *TFileOpenDialog::Execute* ausgeführt. Es gibt nur zwei mögliche Rückgabewerte für *TFileOpenDialog*: IDOK und IDCANCEL. Der zurückgegebene Wert hängt davon ab, ob der Benutzer den »OK«- oder »Abbruch«-Schalter im Dialogfenster für »Datei öffnen« drückt.

Wenn der Rückgabewert IDOK ist ruft *CmFileOpen* die Funktion *OpenFile* auf, welche das Öffnen der Datei durchführt. Die Funktion *Execute* speichert zudem den vom Benutzer ausgewählten Dateinamen in das Element *FileName* von *FileData*. Wenn der Rückgabewert nicht IDOK ist (das heißt, wenn der Rückgabewert IDCANCEL ist), findet keine weitere Aktion statt und die Funktion kehrt zurück.

Die Funktion *CmFileOpen* sollte in etwa wie folgt aussehen:

```
void TMyWindow::CmFileOpen()
{
  if (CanClose())
    if (TFileOpenDialog(this, *FileData).Execute() == IDOK)
      OpenFile();
}
```

Die Funktion CmFileSaveAs

Die Funktion *CmFileSaveAs* kann auf zwei Arten benutzt werden: um eine neue Zeichnung unter einem neuen Namen zu speichern und um eine existierende Zeichnung unter einem anderen als den aktuellen Namen zu speichern.

Um zu bestimmen, welche dieser beiden Aktionen für den Benutzer erforderlich ist, überprüft die Funktion *CmFileSaveAs* zuerst das Flag *IsNewFile*. Wenn die Datei neu ist, kopiert *CmFileSaveAs* einen leeren String in das Element *FileName* von *FileData*. Wenn die Datei nicht neu ist, bleibt *FileName* wie es ist.

Die Unterscheidung zwischen diesen beiden Arten ist sehr wichtig. Wenn *FileName* einen leeren String enthält, ist der vorgegebene Name im Dateinamensfeld der Dialogfensters für Datei öffnen auf den Namensfilter gesetzt, der im *FileData*-Objekt gefunden wurde (in diesem Falle *.pts). Wenn allerdings *FileName* bereits einen Namen enthält, wird dieser Name zusammen mit seinem Verzeichnispfad im Dateinamensfeld eingefügt.

Danach wird *TFileSaveDialog* erzeugt und ausgeführt. Diese Klasse arbeitet genauso wie *TFileOpenDialog* in der Funktion *CmFileOpen*. Wenn die Funktion *Execute* IDOK zurückgibt, ruft *CmFileSaveAs* die Funktion *SaveFile* auf.

Die Funktion *CmFileSaveAs* sollte in etwa wie folgt aussehen:

```
void TMyWindow::CmFileSaveAs()
{
  if (IsNewFile)
    strcpy(FileData->FileName, "");

  if ((new TFileSaveDialog(this, *FileData))->Execute() == IDOK)
    SaveFile();
}
```

Öffnen und Speichern von Zeichnungen

Die Funktionen *CmFileOpen*, *CmFileSave* und *CmFileSaveAs* bieten nur eine Oberfläche, damit der Benutzer Zeichnungen öffnen und speichern kann. Die eigentliche Arbeit zum Öffnen und Speichern von Dateien wird von den Funktionen *OpenFile* und *SaveFile* übernommen. Dieser Abschnitt beschreibt, wie diese Funktionen diese Aktionen durchführen. Er bietet allerdings keine technischen Erklärungen der gesamten Funktionen.

Die Funktion OpenFile

Die Funktion *OpenFile* öffnet die im Element *FileName* des *FileData*-Objektes benannte Datei als ein *ifstream*, einem der C++ Standard-iostreams. Wenn die Datei aus irgendwelchen Gründen nicht geöffnet werden kann, zeigt *OpenFile* ein Meldungsfenster an, um den Benutzer darüber zu informieren, und kehrt zurück.

Wenn die Datei erfolgreich geöffnet wurde, wird das Array *Line* geleert. *OpenFile* liest dann die Zahl der in der Datei gespeicherten Punkte. Diese Zahl ist der erste in der Datei gespeicherte Eintrag. Sie startet dann eine **for**-Schleife, die jeden Punkt in ein temporäres *TPoint*-Objekt liest. Dieses Objekt wird dann in dem Array *Line* hinzugefügt.

Wenn alle Punkte eingelesen wurden, ruft *OpenFile* die Funktion *Invalidate* auf. Diese macht den Fensterbereich ungültig, wodurch eine WM_PAINT-Botschaft gesendet und die neue Zeichnung im Fenster gezeichnet wird.

Zuletzt setzt *OpenFile* die beiden Flags *IsDirty* und *IsNewFile* auf FALSE. Die Funktion *OpenFile* sollte in etwa wie folgt aussehen:

```
void TMyWindow::OpenFile()
{
  ifstream is(FileData->FileName);

  if (!is)
    MessageBox("Unable to open file", "File Error", MB_OK |
MB_ICONEXCLAMATION);
  else {
    Line->Flush();
    unsigned numPoints;
    is >> numPoints;
    while (numPoints--) {
      TPoint  point;
      is >> point;
      Line->Add(point);
    }
  }

  IsNewFile = IsDirty = FALSE;
  Invalidate();
}
```

Die Funktion SaveFile

Die Funktion *SaveFile* öffnet eine im Element *FileName* von *FileData* benannte Datei als ein *ofstream*, einen der C++ Standard-iostreams. Wenn die Datei aus irgendwelchen Gründen nicht geöffnet werden kann, zeigt *SaveFile* ein Meldungsfenster an, um den Benutzer darüber zu informieren, und kehrt zurück.

Wenn die Datei geöffnet wurde, wird die Funktion *Line* **->** *GetItemsInContainer* aufgerufen. Das Ergebnis wird in die Datei eingefügt. Dies Zahl wird durch die Funktion *OpenFile* gelesen, um zu bestimmen, wie viele Punkte in der Datei gespeichert sind.

Danach richtet *SaveFile* einen Iterator mit Namen *i* für *Line* ein. Dieser Iterator durchläuft alle Punkte in dem Array *Line*. Alle Punkte werden dann nacheinander in den Stream eingefügt.

Zuletzt werden *IsNewFile* und *IsDirty* auf FALSE gesetzt. Die Funktion *SaveFile* sollte nun wie folgt aussehen:

```
void TMyWindow::SaveFile()
{
```

```
  ofstream os(FileData->FileName);

  if (!os)
    MessageBox("Unable to open file", "File Error", MB_OK |
MB_ICONEXCLAMATION);
  else {
    os << Line->GetItemsInContainer();
    TPointsIterator i(*Line);
    while (i)
      os << i++;
    IsNewFile = IsDirty = FALSE;
  }
}
```

Die Funktion CmAbout

Die Funktion *CmAbout* verdeutlicht, wie einfach die Verwendung von allgemeinen Dialogfenstern in ObjectWindows ist. Diese Funktion enthält nur eine einzige Zeile Code. Sie benutzt die Klasse *TDialog* und die Dialogfenster-Ressource IDD_ABOUT, um ein Informations-Dialogfenster anzuzeigen. *TDialog* kann bis zu drei Parameter erwarten:

- Der erste Parameter ist ein Zeiger auf das zum Dialogfenster übergeordnete Fenster. Wie bei den Konstruktoren *TFileOpenDialog* und *TFileSaveDialog* können Sie auch hier den **this**-Zeiger benutzen, um das übergeordnete Fenster auf ein *TMyWindow*-Objekt zu setzen.
- Der zweite Parameter ist eine Referenz auf ein *TResId*-Objekt. Dieses sollte der Ressourcen-Bezeichner der Dialogfenster-Ressource sein. Normalerweise übergeben Sie nicht wirklich eine *TResId*-Referenz. Stattdessen übergeben Sie eine Nummer oder einen String für einen Ressourcen-Bezeichner, wie es auch für ein Dialogfenster erfolgt, das über reguläre Windows-API-Aufrufe erzeugt wird. Konvertierungsoperatoren in der Klasse *TResId* lösen den Parameter in den geeigneten Typ auf.
- Der dritte Parameter ist ein *TModule* * und greift normalerweise auf seinen Vorgabewert zurück.

Wenn das Dialogfensterobjekt konstruiert wurde, muß nur noch die Funktion *Execute* aufgerufen werden. Wenn der Benutzer Dialogfenster schließt und die Ausführung beendet ist, kehrt *CmAbout* zurück. Das temporäre *TDialog*-Objekt verläßt den Gültigkeitsbereich und geht verloren.

Der Code für *CmAbout* sollte wie folgt aussehen:

```
void TMyWindow::CmAbout()
{
  TDialog(this, IDD_ABOUT).Execute();
}
```

Weiterführende Informationen

- Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:
- Dialogfenster, einschließlich der Klassen *TFileOpenDialog* und *TFileOpenDialog* sind in Kapitel 10 beschrieben.

Die Funktion *CanClose* ist in Kapitel 3 beschrieben.

Schritt 8: Hinzufügen mehrerer Linien

Sie können den Quelltext für Schritt 8 in den Dateien STEP08.CPP und STEP08.RC im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

Schritt 8 unternimmt nun einen großen Sprung in Richtung auf die Verwendbarkeit der Anwendung. In diesem Schritt werden Sie eine neue Klasse mit Namen *TLine* hinzufügen, die vom *TPoints*-Array abgeleitet ist, welches Sie für die Aufnahme der Punkte in der Linie verwendet haben. Sie werden dann eine andere Array-Klasse *TLines* definieren, die ein Array mit *TLine*-Objekten enthält. Damit können wir mehrere Linien in einem Fenster darstellen. Sie werden Streaming-Operatoren hinzufügen, um die Speicherung der Zeichnungen zu vereinfachen. Zuletzt werden Sie die Funktion *Paint* weiterentwickeln, um Zeichnungen mit mehreren Linien zu verarbeiten.

Die Klasse TLine

Die Klasse *TLine* ist von der öffentlichen Klasse *TPoints* abgeleitet. Diese verleiht *TLine* all die Funktionalität, die Sie mit dem Element *Line* der Klasse *TMyWindow* bereits benutzt haben. Die beinhaltet die bereits verwendeten Funktionen *Add*, *Flush* und *GetItemsInContainer*. Zusätzlich können Sie *TPointsIterator* weiterhin mit der Klasse *TLine* benutzen, wie Sie sie schon mit *TPoints* benutzt haben.

Da Sie aber jetzt Ihre eigene Klasse erzeugen, können Sie ferner jede beliebige Funktionalität, die Sie benötigen, hinzufügen. Sie sollten zum Beispiel ein Datenelement hinzufügen, das die Stiftgröße für jede Linie enthält. Ebenso sollten Sie die Daten nach außen verstecken und ein Manipulieren der Daten nur über zusätzliche Zugriffsfunktionen erlauben.

In *TLine* ist die Stiftgröße in einem **protected int** mit Namen *PenSize* enthalten. Auf *PenSize* wird durch eine von zwei Funktionen zugegriffen. Beide heißen *QueryPen* und beide Versionen geben einen **int**-Wert zurück, welcher den Wert von *PenSize* enthält. Hier ist nun der Unterschied zwischen den beiden Funktionen:

- Die erste *QueryPen*-Funktion erwartet keine Parameter. Diese Funktion gibt die Stiftgröße zurück.
- Die zweite *QueryPen*-Funktion erwartet einen **int**-Wert als einzelnen Parameter. Diese Funktion setzt *PenSize* auf den übergebenen Wert und gibt dann den neuen Wert von *PenSize* zurück. Sie können den Rückga-

bewert benutzen, um zu überprüfen, ob *QueryPen* den Stift tatsächlich auf den von Ihnen übergebenen Wert gesetzt hat. Diese Version von *QueryPen* überprüft den Parameterwert, um sicherzustellen, daß es sich um einen legalen Wert für die Stiftgröße handelt.

TLine enthält ferner eine Definition für den Operator **==**. Dieser Operator prüft, ob die beiden Objekte tatsächlich dieselben Objekte sind. Trifft dies zu, gibt der Operator TRUE zurück. Wenn Sie ein Array mit der Klasse *TArray* definieren (was Sie später bei der Definition von *TLines* tun werden), müssen die im *TArray* verwendeten Objekte einen definierten Operator **==** besitzen.

Zuletzt sollten Sie zwei Operatoren **<<** und **>>** als Friend der Klasse *TLine* deklarieren. Wenn diese Operatoren später in diesem Abschnitt implementiert sind, werden sie einen leichten Zugriff auf Stream-Operationen für die Funktionen *SaveFile* und *OpenFile* bieten.

Hier ist die Deklaration der Klasse *TLine*:

```
class TLine : public TPoints
{
  public:
    TLine(int penSize = 1) : TPoints(10, 0, 10) { PenSize = penSize; }

    int QueryPen() const { return PenSize; }
    int QueryPen(int penSize);

    // Der Operator == muß für die Containerklasse definiert sein,
    // auch wenn er nicht benutzt wird.
    BOOL operator ==(const TLine& other) const
      { return &other == this; }
    friend ostream& operator <<(ostream& os, const TLine& line);
    friend istream& operator >>(istream& is, TLine& line);

  protected:
    int PenSize;
};
```

Das Array TLines

Wenn Sie die Klasse *TLine* definiert haben, können Sie das *TLines*-Array und das *TLinesIterator*-Array definieren. Diese Container arbeiten in gleicher Weise wie die Containerklassen *TPoints* und *TPointsIterator*, die Sie früher definiert haben. Der einzige Unterschied besteht darin, daß *TLines* kein Array mit *TPoint*-Objekten wie *TPoints* enthält, sondern ein Array mit *TLine*-Objekten.

Hier sind die Definitionen von *TLines* und *TLinesIterator*:

```
typedef TArray<TLine> TLines;
typedef TArrayIterator<TLine> TLinesIterator;
```

Einfügen und Extrahieren von TLine-Objekten

Die meisten Objekte, die in eine Datei gesichert und von ihr wiederhergestellt werden müssen, sind auf eine Verwendung von Operatoren zum Einfügen und Extrahieren (Inserter und Extractor) **<<** und **>>** eingerichtet. Durch ein Deklarieren dieser Operatoren als Friends von *TLine* müssen Sie die Operatoren so definieren, daß sie den speziellen in *TLine* gekapselten Datentyp behandeln können.

Mit der Definition dieser Operatoren haben Sie die Möglichkeit, ein vollständiges *TLine*-Objekt mit einer einzigen Code-Zeile in eine Datei zu plazieren. Sie werden sehen, wie dies benutzt wird, wenn Sie die Änderungen an den Funktionen *OpenFile* und *SaveFile* vornehmen.

Der Einfügeoperator <<

Im wesentlichen übernimmt der Einfügeoperator die Funktionalität der in Schritt 7 verwendeten Funktion *SaveFile*. Er muß eine Datei nicht öffnen (was durch diejenige Funktion erledigt wird, die den Operator benutzt) und er besitzt zum Einfügen einen extra Datenwert (*PenSize*). Alles andere unterscheidet sich nicht besonders. Vergleichen Sie die Definition dieser Funktion mit der Funktion *SaveFile* aus Schritt 7. Beachten Sie die Verwendung von *TPointsIterator* mit dem *TLine*-Objekt:

```
ostream& operator <<(ostream& os, const TLine& line)
{
  // Schreibe die Zahl von Punkten in der Linie
  os << line.GetItemsInContainer() << '\n';

  // Schreibe die Stiftgröße
  os << ' ' << line.PenSize;

  // Erhalte einen Iterator für das Array mit Punkten
  TPointsIterator j(line);

  // Solange der Iterator gültig ist (d.h. es stehen noch Punkte aus)
  while(j)
    // Schreibe den Punkt vom Iterator und inkrementiere das Array.
    os << j++;

  os << '\n';

  // Gib das Stream-Objekt zurück
  return os;
}
```

Ausleseoperator >>

>> Operator>>Genauso wie der Einfügeoperator übernimmt auch der Ausleseoperator die Funktionalität der Funktion *OpenFile* aus Schritt 7. Er muß nicht selbst eine Datei öffnen und besitzt einen extra Datenwert zum Auslesen. Alles andere ist bei ihm ähnlich wie bei der Funktion *OpenFile* definiert:

```
istream& operator >>(istream& is, TLine& line)
{
  unsigned numPoints;

  is >> numPoints;

  is >> line.PenSize;

  while (numPoints--) {
    TPoint point;
    is >> point;
    line.Add(point);
  }
  // Gib das Stream-Objekt zurück
  return is;
}
```

Erweitern von TMyWindow

Um eine Anpassung an die neue Klasse *TLine* vorzunehmen, müssen in *TMyWindow* eine Reihe von Änderungen vorgenommen werden. Als erstes sind da eine Reihe von Änderungen bei den Datenelementen:

- *PenSize* wird entfernt. Jede individuelle Linie enthält nun selbst ihre Stiftgröße.
- Das Datenelement *Line* wird von *TPoints* * in *TLine* * geändert. Das *Line*-Objekt enthält die Punkte in der aktuell gezeichneten Linie.
- Das Datenelement *Lines*, ein *TLines* *, wird hinzugefügt. Das *Lines*-Objekt enthält alle *TLine*-Objekte.

Ferner gibt es eine Reihe von Funktionen, die geändert oder hinzugefügt werden:

- Die Funktion *SetPenSize* wird unter **protected** eingerichtet, da Änderungen an der Stiftgröße an der Klasse *TLine* vorgenommen werden sollten. *SetPenSize* sollte nun nur noch intern durch die Klasse *TMyWindow* verwendet werden. *SetPenSize* setzt zudem die Stiftgröße der aktuellen Linie durch den Aufruf der Funktion *QueryPen* dieser Linie.
- Die Funktion *GetPenSize* wird hinzugefügt. Diese Funktion implementiert den *TInputDialog*, der in *EvRButtonDown* behandelt wurde. Der Grund dafür liegt darin, daß zwei Funktionen nun dasselbe Dialogfenster verwenden, *EvRButtonDown* und *CmPenSize*.
- Die Funktion *EvRButtonDown* ruft nun *GetPenSize* auf, um das Eingabedialogfenster zu öffnen.
- Die Funktion *CmPenSize* behandelt das CM_PENSIZE-Ereignis. Dieses Ereignis stammt von einer neuen Menüauswahl, *Stiftgröße* unter einem neuen Menü *Werkzeuge*. Diese Funktion wird hinzugefügt, damit der Benutzer eine andere Möglichkeit zum Ändern der Stiftgröße besitzt.
- Die Funktionen *OpenFile* und *SaveFile* werden geändert, um ein Array mit *TLine*-Objekten anstatt ein Array mit *TPoint*-Objekten zu speichern. Durch die Verwendung der Einfüge- und Ausleseoperatoren ändern

sich diese Funktionen nur sehr wenig gegenüber ihren vorherigen Formen.

Zusätzlich wird die Funktion *Paint* ein wenig verändert. Dies wird im folgenden Abschnitt beschrieben.

Die Funktion Paint

Die Funktion *Paint* muß nun zwei Iterationen an Stelle einer ausführen. Anstatt durch ein einzelnes Array mit Punkten zu iterieren, muß *Paint* nun durch ein Array mit Linien iterieren. Für jede Linie muß sie die Stiftbreite setzen und dann durch die Punkte iterieren, aus denen sich die Linie zusammensetzt.

Dazu erzeugt *Paint* zuerst einen Iterator von *Lines*. Dieser Iterator durchwandert das Array mit Linien. Für jede Linie fragt *Paint* die Stiftgröße der aktuellen Linie ab. Sie setzt das Element *Pen* des Fensters auf diese Größe und wählt diesen Stift für den Gerätekontext aus. Sie erzeugt dann einen Iterator für die aktuelle Linie und inkrementiert den Linien-Array-Iterator.

Der nächste Teil von *Paint* sieht wie die Funktion *Paint* aus Schritt 7 aus. Sie führt nämlich in diesem Teil im wesentlichen dieselben Aktionen wie die vorherige Version von *Paint* aus – sie erwartet ein Array mit Punkten und zeichnet die Linie im Fenster.

Hier ist der Code für die neue Funktion *Paint*:

```
void TMyWindow::Paint(TDC& dc, BOOL, TRect&)
{
  // Iteriere durch das Array mit Linienobjekten.
  TLinesIterator i(*Lines);

  while (i) {
    // Setze den Stift für den dc auf den Stift der aktuellen Linie.
    TPen pen(TColor::Black, i.Current().QueryPen());
    dc.SelectObject(pen);

    // Iteriere durch die Punkte in der Linie i.
    TPointsIterator j(i++);
    BOOL first = TRUE;

    while (j) {
      TPoint p = j++;

      if (!first)
        dc.LineTo(p);
      else {
        dc.MoveTo(p);

        first = FALSE;
      }
    }
  }
}
```

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Fensterklassen sind in Kapitel 8 beschrieben.
- Die Containerklassenbibliothek von Borland und die Klassen *TArray* und *TArrayIterator* sind in Kapitel 9 des Borland C++ *Programmierhandbuches* erklärt.

Schritt 9: Ändern von Stiften

In Schritt 9 werden Sie ein *TColor*-Element für die Klasse *TLine* hinzufügen. Damit kann der Benutzer Linien mit unterschiedlichen Breiten *und* mit unterschiedlichen Farben ziehen. Um die Farbe der Linie zu ändern, werden Sie die Funktion *CmPenColor* hinzufügen. Diese Funktion behandelt das Menükommando CM_PENCOLOR. *CmPenColor* verwendet die Klasse *TChooseColorDialog*, damit der Benutzer die Farben ändern kann. Es werden außerdem einige Hilfsfunktionen hinzugefügt, die das Ändern der Breite und der Farbe unterstützen und die externen Klassen einen Zugriff auf Informationen über die Linie ermöglichen.

Sie können den Quelltext für Schritt 9 in den Dateien STEP09.CPP und STEP09.RC im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

Verbunden mit der Erweiterung des Stiftes um Farbwerte wird in Schritt 9 auch eine neue Funktionalität für die Streaming-Operatoren hinzugefügt, damit die neuen Attribute der Klasse *TLine* berücksichtigt werden. Eine weitere Neuerung ist die Funktion *Draw* für die Klasse *TLine*, um die Klasse besser zu strukturieren und die Funktion *Paint* einfacher zu gestalten.

Änderungen an der Klasse TLine

Eine Reihe von Änderungen an der Klassendeklaration von *TLine* sind erforderlich, um die neue Funktionalität zu erreichen:

- Es gibt ein neues Datenelement unter **protected** mit Namen *Color* (ein *TColor*-Objekt). *Color* und *PenSize* bilden zusammen die Attribute, die für das Konstruieren eines *TPen*-Objektes notwendig sind.
- Der Kopf des Konstruktors ändert sich von:

```
TLine(int penSize = 1);
```

in:

```
TLine(const TColor &color = (TColor) 0, int penSize = 1);
```

Der Konstruktor selbst ändert sich dahingehend, daß *PenSize* als zweiter Parameter des Konstruktors gesetzt wird und daß ein neues *TPen*-Objekt erzeugt und an *Pen* zugewiesen wird. Wenn keine Parameter angegeben sind und der erste Parameter seinen Vorgabewert annimmt, wird *TColor::Black* als Stiftfarbe benutzt.

- Die zwei *QueryPen*-Funktionen werden zugunsten von drei neuen Funktionen aufgegeben: *QueryPenSize* gibt die Stiftgröße als ein **int** zurück, *QueryColor* gibt die Stiftfarbe als ein *TColor* zurück und *QueryPen* gibt den Stift als ein *TPen* zurück.
- An Stelle von Abfragefunktionen für das Setzen der Stiftattribute gibt es zwei neue Funktionen mit Namen *SetPen*. Eine erwartet einen **int**-Wert als einzigen Parameter und die andere erwartet ein *TColor* **&** und zwei **int**-Werte. Die Funktionen zur Abfrage und zum Setzen des Stiftes werden im nächsten Abschnitt beschrieben.
- Eine Funktion *Draw* wird hinzugefügt, so daß die Klasse *TLine* diktiert, wie sie gezeichnet wird. Diese Funktion ist **virtual**, so daß sie leicht in einer abgeleiteten Klasse überschrieben werden kann.

Hier ist nun ein Beispiel, wie die neue Klassendeklaration von *TLine* aussehen sollte:

```
class TLine : public TPoints {
  public:
    // Konstruktor für das Konstruieren mit einer Farbe und einer
    // Stiftgröße. Dient auch als Standardkonstruktor.
    TLine(const TColor &color = TColor(0), int penSize = 1)
      : TPoints(10, 0, 10), PenSize(penSize), Color(color) {}

    // Funktionen für das Ändern und Abfragen von Stiftattributen.
    int QueryPenSize() { return PenSize; }
    TColor& QueryColor() { return Color; }
    void SetPen(TColor &newColor, int penSize = 0);
    void SetPen(int penSize);

    // TLine zeichnet sich selbst. Gibt TRUE zurück, wenn alles
    // korrekt ablief.
    virtual BOOL Draw(TDC &) const;

    // Der Operator == muß für die Containerklasse definiert sein,
    // auch wenn er nicht benutzt wird.
    BOOL operator ==(const TLine& other) const
      { return &other == this; }
    friend ostream& operator <<(ostream& os, const TLine& line);
    friend istream& operator >>(istream& is, TLine& line);

  protected:
    int PenSize;
    TColor Color;
};
```

Stift-Zugriffsfunktionen

In Schritt 8, konnte die Funktion *QueryPen* sowohl für den Zugriff auf die aktuelle Größe des Stiftes wie auch für das Setzen der Stiftgröße verwendet werden. Die neuen Abfragefunktionen von *TLine* – *QueryPenSize* und *QueryColor* – können für das Ändern der Stiftattribute nicht verwendet werden. Diese Funktionen geben Stiftattribute nur zurück.

Um die Stiftattribute zu setzen, gibt es zwei neue Funktionen mit Namen *SetPen*. Die erste Version von *SetPen* setzt nur die Stiftgröße. Die andere Version von *SetPen* kann für das Setzen der Farbe, der Größe oder des Stiles für den Stift verwendet werden. Wenn Sie aber den zweiten und dritten Parameter jeweils bei seinem Vorgabewert belassen, können Sie den zweiten Konstruktor verwenden, um nur die Farbe zu setzen. Hier ist der Code für diese Funktionen:

```
void TLine::SetPen(int penSize)
{
  if (penSize < 1)
    PenSize = 1;
  else
    PenSize = penSize;
}

void TLine::SetPen(TColor &newColor, int penSize)
{
  // Wenn penSize nicht der Vorgabewert (0) ist, setze PenSize
  // auf die neue Größe.
  if (penSize)
    PenSize = penSize;

  Color = newColor;
}
```

Die Funktion Draw

Die Funktion *Draw* zeichnet die Linie im Fenster, wobei sie die Funktionalität von der Funktion *Paint* des Fensters übernimmt. Diese Funktionalität wird verschoben, da das *TLine*-Objekt nun diktieren kann, wie es auf dem Bildschirm zu zeichnen ist. Betrachten Sie den unten stehenden Code für die Funktion *Draw* und vergleichen Sie ihn mit der Funktion *Paint* aus Schritt 8. Aus einer gewissen Sichtweise sind die beiden Code-Fragmente identisch:

```
BOOL TLine::Draw(TDC &dc) const
{
  // Setze den Stift für den dc auf die Werte für diese Linie
  TPen pen(Color, PenSize);
  dc.SelectObject(pen);

  // Iteriere durch die Punkte in der Linie i.
  TPointsIterator j(*this);
  BOOL first = TRUE;

  while (j) {
    TPoint p = j++;

    if (!first)
      dc.LineTo(p);
    else {
      dc.MoveTo(p);
      first = FALSE;
```

```
    }
  }
  dc.RestorePen();
  return TRUE;
}
```

Nachdem dieser gesamte Code in die Klasse *TLine* verlegt ist, wird die Funktion *TMyWindow::Paint* erheblich vereinfacht:

```
void TMyWindow::Paint(TDC& dc, BOOL, TRect&)
{
  // Iteriere durch das Array mit Linienobjekten.
  TLinesIterator i(*Lines);

  while (i)
    i++.Draw(dc);
}
```

Einfüge- und Ausleseoperatoren

Es gibt weiterhin einige Änderungen an den Einfüge- und Ausleseoperatoren, die für das Behandeln der überarbeiteten Klasse *TLine* notwendig sind.

Der Einfügeoperator wird modifiziert, um das Element *PenSize* und *Color* hinauszuschreiben. Er schreibt danach die Punkte genauso wie schon zuvor hinaus.

Der Ausleseoperator liest die Daten ein und benutzt die Daten *PenSize* und *Color* in der Funktion *SetPen*. Jeder Punkt wird von der Datei eingelesen und dem Objekt hinzugefügt.

Änderungen an der Klasse TMyWindow

Es gibt einige geringfügige Änderungen an der Klasse *TMyWindow*, die für die Anpassung an die überarbeitete Klasse *TLine* notwendig sind:

- Das Datenelement *Pen* wird von der Größe und Farbe der aktuellen Linie aus konstruiert.
- Die Funktion *SetPenSize* wird entfernt. Die Funktion *GetPenSize* öffnet ein *TInputDialog* für den Benutzer, damit er die neue Stiftgröße eingeben kann. *GetPenSize* ruft dann die Funktion *Lin* **->**SetPen auf, um die Stiftgröße wirklich zu setzen.
- Die Funktion *CmPenColor* wird hinzugefügt, um das Ereignis CM_PENCOLOR zu behandeln. Dieses Ereignis wird von der neuen *Werkzeuge*-Menüauswahl *Stiftfarbe* gesendet.

Die Funktion CmPenColor

Die Funktion *CmPenColor* öffnet ein *TChooseColorDialog* für den Benutzer, damit er eine Farbe auswählen kann. Wie *TFileOpenDialog* und *TFileSaveDialog* ist *TChooseColorDialog* eine Kapselung eines allgemeinen Dialogfensters von Windows.

Genauso wie *TFileOpenDialog* und *TFileSaveDialog* kann der Konstruktor *TChooseColorDialog* bis zu fünf Parameter erwarten. In dem hier beschriebenen Fall benötigen Sie jedoch nur zwei. Die letzten drei Parameter nehmen Vorgabewerte an. Die zwei benötigten Parameter sind ein Zeiger auf das übergeordnete Fenster und eine Referenz auf ein *TChooseColorDialog::TData*-Objekt. In diesem Fall, ist der Zeiger auf das übergeordnete Fenster einfach nur der **this**-Zeiger. Das *TChooseColorDialog::TData*-Objekt wird durch *colors* bereitgestellt.

Das Setzen des Elementes *Color* von *colors* auf eine bestimmte Farbe läßt diese Farbe (oder da beste Äquivalent, das im Dialogfenster angezeigt wird) zur Standardfarbe im Dialogfenster werden. Indem Sie *Color* auf die Farbe des aktuellen Stiftes setzen, stellen Sie sicher, daß das Farb-Dialogfenster den aktuellen Zustand der Anwendung reflektiert.

Das Setzen des Elementes *CustColors* des *colors*-Objektes auf ein Array mit *TColor*-Objekten setzt diese Farben im Abschnitt mit selbstdefinierten Farben des Farb-Dialogfensters. Sie können für das Array *CustColors* beliebige Farben auswählen. Die in dieser Einführung verwendeten Werte erzeugen einen Bereich von Monochromfarben, der sich von Schwarz bis Weiß erstreckt.

Das Erzeugen und Ausführen eines *TChooseColorDialog* arbeitet exakt genauso wie bei *TFileOpenDialog* oder *TFileSaveDialog*. Obwohl das Farb-Dialogfenster einen extra Schalter besitzt (den Schalter zum Definieren eigener Farben) wird dieser Schalter durch den Windows-Bestandteil des allgemeinen Dialogfensters behandelt. Deshalb gibt es nur zwei möglicher Ergebnisse für die Funktion *Execute*, nämlich IDOK und IDCANCEL. Wenn der Benutzer »Abbruch« ausgewählt hat, ignorieren Sie alle Änderungen im Dialogfenster.

Wenn der Benutzer andererseits »OK« auswählt, müssen Sie die Stiftfarbe auf den neuen, durch den Benutzer ausgewählten Farbwert setzen. *TChooseColorDialog* plaziert die durch den Benutzer ausgewählte Farbe in das Element *Color* des *colors*-Objektes. *Color* ist ein *TColor*, welches sehr gut in die Funktion *SetPen* eines *TLine*-Objektes paßt.

Hier ist der Code für die Funktion *CmPenColor*:

```
void TMyWindow::CmPenColor()
{
  TChooseColorDialog::TData colors;
  static TColor custColors[16] =
  {
    0x010101L, 0x101010L, 0x202020L, 0x303030L,
    0x404040L, 0x505050L, 0x606060L, 0x707070L,
    0x808080L, 0x909090L, 0xA0A0A0L, 0xB0B0B0L,
    0xC0C0C0L, 0xD0D0D0L, 0xE0E0E0L, 0xF0F0F0L
  };
```

```
    colors.Flags = CC_RGBINIT;
    colors.Color = TColor(Line->QueryColor());
    colors.CustColors = custColors;
    if (TChooseColorDialog(this, colors).Execute() == IDOK)
      Line->SetPen(colors.Color);
}
```

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Die Klassen *TPen* und *TColor* sind in Kapitel 15 beschrieben.
- Dialogfenster wie auch die Klasse *TChooseColorDialog* sind in Kapitel 10 beschreiben.

Schritt 10: Hinzufügen von Dekorationen

Die einzigen Änderungen in Schritt 10 betreffen die Funktion *InitMainWindow*. Allerdings bewirken diese Änderungen einen eleganteren, einfacheren und mehr intuitiven Einsatz Ihrer Anwendung. In diesem Schritt werden Sie eine Dialogelementleiste mit Bitmap-Schalter-Gadgets und eine Statuszeile, die die aktuelle Menüauswahl anzeigt, hinzufügen.

Dieser Schritt besteht aus vier wesentlichen Änderungen:

Sie können den Quelltext für Schritt 10 in den Dateien STEP10.CPP und STEP10.RC im Verzeichnis EXAMPLES\OWL\TUTORIAL finden.

- Ändern des Hauptfensters von *TFrameWindow* in *TDecoratedFrame*.
- Erzeugen einer Statuszeile und Einfügen dieser Zeile in das dekorierte Rahmenfenster.
- Erzeugen einer Dialogelementleiste zusammen mit ihren Schalter-Gadgets und Einfügen dieser Leiste in das dekorierte Rahmenfenster.
- Hinzufügen von Ressourcen, wie zum Beispiel eine String-Tabelle (die Beschreibungen für jede der verfügbaren Menüauswahlen bereitstellt) und Bitmaps für die Schalter-Gadgets.

Ändern des Hauptfensters

Eine Änderung von *TFrameWindow* in *TDecoratedFrame* ist recht einfach. Da *TDecoratedFrame* auf *TFrameWindow* basiert, kann ein dekorierter Rahmen überall da verwendet werden, wo bereits ein regulärer Rahmen vorliegt. In diesem Fall erzeugen Sie nur einen *TDecoratedFrame* und übergeben ihn als Parameter an die Funktion *SetMainWindow*.

Selbst die Konstruktoren von *TFrameWindow* und *TDecoratedFrame* sind ähnlich. Der einzige Unterschied liegt beim vierten Parameter, welcher ohnehin nicht benutzt wurde. Für *TFrameWindow* ist der vierte Parameter ein BOOL, der dem Rahmenfenster mitteilt, ob es auf die Größe seines Client-Fensters schrumpfen soll.

Für *TDecoratedFrame* ist der vierte Parameter ebenfalls ein BOOL. Dieser Parameter zeigt an, ob der dekorierte Rahmen Menüauswahlen mitverfolgen soll. Bei der Mitverfolgung des Menüs wird eine Textbeschreibung zum aktuell ausgewählten Menüeintrag oder Schalter in einer Meldungszeile oder Statuszeile angezeigt. Wenn Sie bei diesem Parameter TRUE angeben, *müssen* Sie eine Meldungs- oder Statuszeile für das Fenster bereitstellen. Wenn Sie dies versäumen, wird Ihre Anwendung abstürzen, sobald sie das erste Mal versucht, eine Meldung an die Meldungs- oder Statuszeile zu senden.

Wenn Sie eine Statuszeile einsetzen, müssen Sie die Ressourcen dafür in Ihrer Ressourcendatei mit einbinden. Diese Ressourcen sind in der Datei STATUSBA.RC im Verzeichnis INCLUDE\OWL enthalten.

Der einzige andere Unterschied besteht darin, daß der dekorierte Rahmen einige Vorbereitungen wie zum Beispiel das Hinzufügen von Dekorationen wie der Dialogelementleiste und der Statuszeile erfordert, bevor er zum Hauptfenster werden kann. Damit können Sie das Fenster nicht mehr in einem Schritt Konstruieren und Einrichten, sondern müssen das Fenster konstruieren, es vorbereiten und es dann als das Hauptfenster setzen.

Erzeugen der Statuszeile

Statuszeilen werden mit Hilfe der Klasse *TStatusBar* erzeugt. *TStatusBar* basiert auf der Klasse *TMessageBar*, die selbst wiederum auf *TGadgetWindow* basiert. Sowohl Meldungszeilen als auch Statuszeilen zeigen Textmeldungen an. Statuszeilen besitzen allerdings mehr Optionen als Meldungszeilen. Sie können zum Beispiel mehrere Text-Gadgets (kleine Felder für Text), Ränder mit verschiedenen Eigenschaften und Modus-Indikatoren (wie zum Beispiel für den Einfüge/Überschreibe-Modus) in einer Statuszeile plazieren.

Der Konstruktor von *TStatusBar* erwartet fünf Parameter, auch wenn Sie hier nur die ersten beiden benutzen. Alle anderen Parameter nehmen ihre Vorgabewerte an:

- Der erste Parameter ist ein Zeiger auf das der Statuszeile übergeordnete Fenster. In diesem Falle verwenden Sie *frame*. Dies ist der Zeiger auf das vorher konstruierte dekorierte Rahmenfenster.
- Der zweite Parameter ist ein *TGadget::Borderstyle* **enum**. Er kann einer der folgenden Werte sein: *None*, *Plain*, *Raised*, *Recessed* oder *Embossed*. Dieser Parameter bestimmt die Eigenschaften der Statuszeile und hat den Vorgabewert *Recessed*.
- Der dritte Parameter ist ein *TModeIndicator* **enum**. Er bestimmt die Tastaturmodi, die die Statuszeile zeigen soll. Diese Indikatoren können einen oder mehrere der folgenden Werte sein: *ExtendSelection*, *CapsLock*, *NumLock*, *ScrollLock*, *Overtype* und *RecordingMacro*. Dieser Parameter hat den Vorgabewert Null. Dies bedeutet, daß keine Tastaturmodi angezeigt werden.

- Der vierte Parameter ist ein *TFont* *****. Er enthält die in der Statuszeile zu verwendende Schrift. Als Vorgabewert dient *TGadgetWindowFont*.
- Der fünfte Parameter ist ein *TModule* *****. Er hat als Vorgabewert Null.

Hier ist der Konstruktor der Statuszeile:

```
TStatusBar* sb = new TStatusBar(frame, TGadget::Recessed);
```

Wenn die Statuszeile erzeugt ist, kann sie in den dekorierten Rahmen eingefügt werden.

Erzeugen der Dialogelementleiste

Das Erzeugen der Dialogelementleiste ist etwas komplizierter als das Erzeugen der Statuszeile. Sie konstruieren zuerst das eigentliche *TControlBar*-Objekt. Dann erzeugen Sie die Gadgets, die die Dialogelemente in der Leiste bilden, und fügen sie in die Dialogelementleiste ein.

Konstruieren von TControlBar

Der Konstruktor *TControlBar* erwartet vier Parameter. Für die hier verfolgten Zwecke benötigen Sie allerdings nur den ersten. Alle anderen Parameter nehmen ihre Vorgabewerte an:

- Der erste Parameter ist ein Zeiger auf das übergeordnete Fenster. Wie bei der Statuszeile verwenden Sie hier *frame*, um den dekorierten Rahmen zum übergeordneten Fenster für die Dialogelementleiste werden zu lassen.
- Der zweite Parameter ist ein *TTileDirection* **enum**. Ein *TTileDirection* **enum** kann zwei Werte annehmen, *Horizontal* und *Vertical*. Dieser Wert teilt der Dialogelementleiste mit, aus welche Art ihre Dialogelemente angeordnet werden. Dieser Parameter hat den Vorgabewert *Horizontal*.
- Der dritte Parameter ist ein *TFont* *****. Er enthält die in der Statuszeile zu verwendende Schrift. Standardmäßig ist dies *TGadgetWindowFont*.
- Der vierte Parameter ist ein *TModule* ***** mit Vorgabewert Null.

Hier ist der Konstruktor der Dialogelementleiste:

```
TControlBar *cb = new TControlBar(frame);
```

Erzeugen von Schalter-Gadgets

Schalter-Gadgets werden als Schalter der Dialogelementleiste benutzt. Sie assoziieren einen Bitmap-Schalter mit einem Ereignisbezeichner. Wenn der Benutzer einen Schalter-Gadget drückt, sendet dieses den Ereignisbezeichner. Sie können dies so einstellen, daß das Drücken des Schalters in der Dialogelementleiste der Auswahl eines Menüeintrages entspricht. In diesem Abschnitt werden Sie sehen, wie Schalter eingerichtet werden, damit sie jedes Ihrer aktuellen Menüeinträge nachbilden.

Schalter-Gadgets werden mit Hilfe der Klasse *TButtonGadget* konstruiert. Der Konstruktor *TButtonGadget* erwartet sechs Parameter, von denen Sie hier nur die ersten drei benötigen:

- Der erste Parameter ist eine Referenz auf ein *TResId*-Objekt (siehe die Bemerkung weiter unten bezüglich der Klasse *TResId*). Dies sollte der Ressourcen-Bezeichner der Bitmap sein, die Sie für den Schalter vorsehen. Bezüglich der Bitmap-Größe gibt es für die Verwendung in einem Schalter-Gadget keine Einschränkungen. Allerdings gibt es einige Punkte in diesem Zusammenhang zu bedenken: Die Höhe der Dialogelementleiste basiert auf der Höhe der Objekte, die in der Dialogelementleiste enthalten sind. Wenn Ihre Bitmap extrem groß ist, wird es auch Dialogelementleiste sein.
- Der zweite Parameter ist der Gadget-Bezeichner für dieses Schalter-Gadget. Normalerweise sind Gadget-Bezeichner, Ereignisbezeichner und Bitmap-Ressourcen-Bezeichner identisch. So verwendet zum Beispiel das Schalter-Gadget für das Kommando *Datei / Öffnen* eine Bitmap-Ressource mit Namen CM_FILEOPEN, besitzt einen Gadget-Bezeichner CM_FILEOPEN und hinterlegt ein CM_FILEOPEN-Ereignis.

  Der Bitmap ist in der Ressourcendatei derselbe Bezeichner wie dem Ereignis vergeben. Dies erleichtert Ihnen die Arbeit mit dem Code. Dies ist jedoch *kein*e Regel und Sie können die Namen für die Bitmap und das Ereignis beliebig anders vergeben. Die einzige Bedingung besteht darin, daß der Ereignisbezeichner definiert sein muß und eine Art von Verarbeitung vorsieht und daß der Ressourcen-Bezeichner gültig sein muß.

  Sie sollten zudem beachten, daß es eine Reihe von Einträgen in der String-Ressourcen-Tabelle der Anwendung gibt, die dieselben IDs wie die Gadgets und Ereignisse besitzen. Wenn ein String mit demselben Bezeichner wie ein Schalter-Gadget existiert, wird dieser String in der Statuszeile beim Drücken des Gadget angezeigt.
- Der dritte Parameter ist ein *TType* **enum**. Er zeigt an, welchen Typ der Schalter besitzt. Drei verschiedene Schaltertypen sind möglich: *Command*, *Exclusive* und *NonExclusive*. In dieser Anwendung, sind alle Schalter Kommandoschalter. Dieser Parameter hat den Vorgabewert *Command*.
- Der vierte Parameter ist ein BOOL-Wert, der anzeigt, ob der Schalter aktiviert ist. Standardmäßig ist dieser Parameter FALSE.
- Der fünfte Parameter ist ein *TState* **enum**. Dieser Parameter gibt den Anfangszustand des Schalters an und kann die folgenden Werte annehmen: *Up*, *Down* oder *Indeterminate*. Dieser Parameter hat den Vorgabewert *Up*.
- Der sechste Parameter ist ein BOOL-Wert, der den Wiederholzustand des Schalters anzeigt. Wenn der Wiederholzustand TRUE ist, wiederholt der Schalter seine Funktion, wenn der gedrückt gehalten wird. Standardmäßig ist dieser Parameter auf FALSE eingestellt.

Separator-Gadgets

Es gibt einen weiteren Gadget-Typ, der beim Konstruieren von Dialogelementleisten oft verwendet wird. Es hat den Namen Separator-Gadget. Normalerweise werden Gadgets in einer Dialogelementleiste direkt rechts neben den anderen plaziert. Ein Separator-Gadget stellt einen kleinen Zwischenraum zwischen zwei Gadgets her. Dadurch können Sie Gadgets in Gruppen auftrennen, sie in vorbestimmte Felder auf der Dialogelementleiste plazieren usw.

Separator-Gadgets sind in der Klasse *TSeparatorGadget* enthalten. Dies ist eine einfache Klasse, die einen einzelnen **int**-Parameter aufnimmt. Standardmäßig ist der Wert dieses Parameters Sechs. Der Parameter zeigt die Zahl von Pixeln für den Zwischenraum an, der durch den Separator-Gadget eingerichtet werden soll.

Einfügen von Gadgets in die Dialogelementleiste

Wenn Sie Ihre Gadgets konstruiert haben, müssen Sie sie in die Dialogelementleiste einfügen. Die Dialogelementleiste kann Gadgets aufnehmen, da sie von der Klasse *TGadgetWindow* abgeleitet ist. *TGadgetWindow* bietet die grundlegende Funktionalität für die Verwendung von Gadgets in einem Fenster. *TControlBar* verfeinert diese Funktionalität zum Erzeugen einer Dialogelementleiste.

Mit Hilfe der Funktion *Insert* können Sie Gadgets in die Dialogelementleiste einfügen. Diese Version der Funktion *Insert* ist durch *TControlBar* von *TGadgetWindow* geerbt (später werden Sie eine andere Version dieser Funktion, die in *TDecoratedFrame* enthalten ist, verwenden). Diese Funktion erwartet drei Parameter, von denen Sie in der Beispielanwendung nur den ersten Parameter benötigen:

- Der erste Parameter ist eine Referenz auf ein *TGadget*-Objekt oder ein von *TGadget* abgeleitetes Objekt.
- Der zweite Parameter ist ein *TPlacement* **enum**, welcher zwei Werte annehmen kann: *Before* oder *After*. Dieser Parameter zeigt an, ob das Gadget vor oder hinter dem zum Gadget gleichgeordneten Objekt plaziert werden soll. Der Vorgabewert ist *After*. Dieser Parameter hat keinen Effekt, wenn kein gleichgeordnetes Objekt angegeben ist.
- Das zum Gadget gleichgeordnete Objekt wird im dritten Parameter angegeben, der ein *TGadget* ***** ist. Das gleichgeordnete Objekt sollte in die Dialogelementleiste bereits eingefügt sein. Der Vorgabewert des Parameters ist Null.

In der Beispielanwendung wird das Konstruieren der Gadgets und das Einfügen in die Dialogelementleiste in einem einzigen Schritt ausgeführt. Hier ist der Code, in dem Gadgets in die Dialogelementleiste eingefügt werden:

```
cb->Insert(*new TButtonGadget(CM_FILENEW, CM_FILENEW,
           TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN,
           TButtonGadget::Command));
```

```
cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE,
           TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_FILESAVEAS, CM_FILESAVEAS,
           TButtonGadget::Command));
cb->Insert(*new TSeparatorGadget);
cb->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE,
           TButtonGadget::Command));
cb->Insert(*new TSeparatorGadget);
cb->Insert(*new TButtonGadget(CM_ABOUT, CM_ABOUT,
           TButtonGadget::Command));
```

Beachten Sie, daß die Schalter-Gadgets die bereits vorhandenen Menükommandos nachbilden. Dadurch erhält der Benutzer die Möglichkeit, auf häufig benutzte Menükommandos einfacher zuzugreifen. Selbstverständlich gibt es keine Beschränkung dafür, Gadgets in einer Dialogelementleiste nur als Ersatz oder Abkürzung für Menükommandos zu verwenden. Mit Hilfe des Parameters *TType* können Sie Gadgets auf einer Dialogelementleiste einrichten, die wie Schaltfelder (indem Sie *Exclusive* mit einer Gruppe von Gadgets verwenden), Markierungsfelder (indem Sie *NonExclusive* verwenden) usw. arbeiten.

Einfügen von Objekten in einen dekorierten Rahmen

Nachdem Sie nun die Dekorationen für Ihr *TDecoratedFrame*-Fenster konstruiert haben, müssen Sie nur noch die Dekorationen in das Fenster einfügen und Das Fenster zum Hauptfenster machen.

Das Einfügen der Dekorationen in einen dekorierten Rahmen ähnelt dem Einfügen von Gadgets in eine Dialogelementleiste. Die Funktion *TDecoratedFrame::Insert* erwartet zwei Parameter:

- Der erste Parameter ist eine Referenz auf ein *TWindow*-Objekt oder ein von *TWindow* abgeleitetes Objekt. Dieses *TWindow*-Objekt ist die Dekoration. In diesem Fall sind die von *TWindow* abgeleiteten Objekte das *TStatusBar*-Objekt und das *TControlBar*-Objekt.
- Der zweite Parameter ist ein *TLocation* **enum**. Dieser Parameter kann einen der vier folgenden Werte annehmen: *Top*, *Bottom*, *Left* oder *Right*. Damit wird angezeigt, wo das Gadget im dekorierten Rahmen plaziert werden soll.

Hier ist der Code für das Einfügen der Dekorationen in den dekorierten Rahmen:

```
// Füge die Statuszeile und die Dialogelementleiste in den Rahmen ein
frame->Insert(*sb, TDecoratedFrame::Bottom);
frame->Insert(*cb, TDecoratedFrame::Top);
```

Wenn Sie die Dekorationen in den Rahmen eingefügt haben, müssen Sie als letztes das Hauptfenster auf *frame* setzen und das Menü einrichten:

```
// Setze das Hauptfenster und sein Menü
SetMainWindow(frame);
GetMainWindow()->AssignMenu("COMMANDS");
```

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

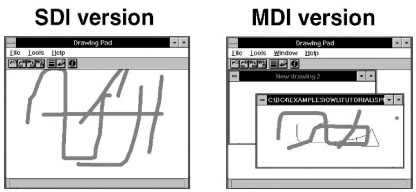
- Dekorierte Rahmen sind in Kapitel 8 beschrieben.
- Statuszeilen und Dialogelementleisten sind in Kapitel 12 beschrieben.
- Gadgets sind in Kapitel 13 beschrieben.

Schritt 11: Übergang zu MDI

Dieses Kapitel beschreibt, wie die in Schritt 10 erzeugte Anwendung für die Benutzung des MDI (Multiple Document Interface) zu konvertieren ist. Die Anwendung aus Schritt 10 bezeichnet man als SDI-Anwendung (Single Document Interface), d.h. die Anwendung kann immer nur ein einzelnes Dokument unterstützen und anzeigen.

In dem Sinn wie *Dokument* hier verwendet wird, hat es vielleicht nicht die Bedeutung, die Sie sonst gewohnt sind. Bei einem Dokument handelt es sich hier nicht um eines aus Papier oder einer Textverarbeitung, sondern vielmehr um jede Datenmenge, die Ihre Anwendung anzeigen oder bearbeiten kann. Im Fall der Beispielanwendung sind Dokumente etwa die Bilddateien, die diese Anwendung erzeugt. Wird eine solche Anwendung für die Benutzung von MDI konvertiert, so kommt die Fähigkeit hinzu, mehrere Abbildungen zur gleichen Zeit in untergeordneten Fenstern geöffnet zu haben. Abbildung 2.1 zeigt den Unterschied zwischen der SDI-Version der Zeichenbrettanwendung und der MDI-Version, die Sie in diesem Schritt erzeugen werden.

Abbildung 2.1: SDI im Vergleich mit MDI bei der Zeichenbrettanwendung



Verstehen des MDI-Modells

Eine MDI-Anwendung funktioniert etwas anders als eine SDI-Anwendung. Die Zeichenbrettanwendung in Schritt 10 zeigte eine Abbildung je Fenster, und das die Abbildung zeigende Fenster war ein Client des Rahmenfensters. Das Rahmenfenster verwaltete allgemeine Tasks der Anwendung, wie die Menühandhabung, die Größenänderung, das Zeichnen von Menüs und Dialogelementleisten usw. Das Client-Fenster verwaltete

anwendungaspezifischen Tasks, wie die Handhabung der Mausbewegungen und Tastenklicks im Client-Bereich, das Zeichnen der Linien in der Abbildung, die Bearbeitung anwendungsspezifischer Ereignisse usw.

Im Vergleich dazu spalten MDI-Anwendungen die Tasks in drei statt in zwei Sorten auf:

- Das Rahmenfenster funktioniert bei der Handhabung des Funktionsumfangs der Anwendung im wesentlichen genauso wie bei SDI-Anwendungen.
- Das Client-Fenster handhabt Tasks, die mit der Erzeugung, Verwaltung und dem Schließen von untergeordneten MDI-Fenstern und ähnlichen Funktionen in Zusammenhang stehen. Beispielsweise verwaltet das Client-Fenster den Befehl *Datei / Öffnen* , da Sie, um ein untergeordnetes MDI-Fenster zu öffnen, im allgemeinen irgendetwas brauchen, worin dies angezeigt werden kann.
- Untergeordnete MDI-Fenster zeigen ihre Daten in einer MDI-Anwendung an und geben dem Benutzer die Möglichkeit, die Daten zu manipulieren und zu bearbeiten. Diese Fenster behandeln anwendungsspezifische Tasks, ähnlich denen des Client-Fenster in Schritt 10.

In diesem Schritt werden Sie das Beispiel aus Schritt 10 nehmen und es zwecks Unterstützung der MDI-Funktionalität umbauen. Dies ist nicht so kompliziert, wie es vielleicht aussehen mag; die meisten der neu zu konstruierenden Klassen lassen sich direkt aus der existierenden Klasse *TDrawWindow* übernehmen.

Hinzufügen der MDI Header-Dateien

Um Ihrer Anwendung die MDI-Fähigkeit hinzuzufügen, müssen Sie einige neue Header-Dateien mit einbinden. Dieser Abschnitt beschreibt die Header-Dateien, die zu ändern oder hinzuzufügen sind, sowie die Klassen, die in den jeweiligen Header-Dateien definiert sind.

Ändern der Ressourcen-Script-Datei

Sie müssen die Include-Anweisung für die Ressourcen-Script-Datei step10.rc ändern, um statt dessen die Ressourcen-Script-Datei step11.rc einzubinden. Es sind lediglich zwei Änderungen für step11.rc notwendig:

- Fügen Sie die Ressourcen-Header-Datei owl\mdi.rh ein.
- Fügen Sie ein Popup-Menü namens Fenster zwischen dem Tools-Menü und dem Hilfe-Menü ein. Dieses Menü sollte die vier in Tabelle 2.1 beschriebenen Einträge haben.

Tabelle 2.1: Einträge und Bezeichner des MDI-Fenstermenüs

Menüeintrag	Befehlsbezeichner
Überlappend	CM_CASCADECHILDREN
Nebeneinander	CM_TILECHILDREN
Symbole anordnen	CM_ARRANGEICONS
Alle schließen	CM_CLOSECHILDREN

Die Funktionen zur Behandlung dieser Ereignisse werden später auf beschrieben.

Ersetzen der Rahmenfenster-Header-Datei

Statt owl\decframe.h müssen Sie owl\decmdifr.h einbinden. Diese Header-Datei enthält die Definition der Klasse *TDecoratedMDIFrame*, die von *TMDIFrame* und von *TDecoratedFrame* abgeleitet ist. Die in der Header-Datei owl\mdi.h definierte Klasse *TMDIFrame* fügt die Unterstützung zum Einbetten eines MDI-Client-Fensters der bereits von *TFrameWindow* gelieferten Unterstützung für die Befehlsverarbeitung und Tastaturverwaltung hinzu. MDI-Client-Fenster werden später betrachtet. Wie im vorangehenden Schritt des Tutorials gezeigt wurde, bietet *TDecoratedFrame* die Unterstützung von Dekorationen, wie etwa Dialogelementleisten und Statuszeilen. Da die Beispielanwendung bereits im vorangegangenen Schritt Dekorationen unterstützte, können Sie die dekorierte Version des MDI-Rahmenfensters verwenden, um diese Fähigkeit beizubehalten.

Hinzufügen von MDI-Client- und untergeordneten Header-Dateien

Sie müssen die Header-Dateien owl\mdi.h und owl\mdichild.h hinzufügen. owl\mdi.h enthält die Definition der Klassen *TMDIFrame* und *TMDIClient*. *TMDIClient* liefert den für die Verwaltung von untergeordneten MDI-Fenstern nötigen Funktionsumfang. Untergeordnete MDI-Fenster sind solche, mit denen der Benutzer Ihrer Anwendung im eigentlichen Sinne arbeitet und worin die in den Dokumenten enthaltenen Daten angezeigt werden. *TMDIClient* bietet die Fähigkeit:

- alle offenen untergeordneten MDI-Fenster zu schließen
- das aktive untergeordnete MDI-Fenster herauszusuchen
- ein neues untergeordnetes MDI-Objekt zu initialisieren
- ein neues untergeordnetes MDI-Fenster zu erzeugen
- untergeordnete MDI-Fenster anzuordnen und zu verwalten, einschließlich der Anordnung ihrer Symbole und überlappender oder nebeneinander liegender offener Fenster.

owl\mdichild.h beinhaltet die Definition der Klasse *TMDIChild*, die von *TWindow* abgeleitet ist. *TMDIChild* überschreibt einige Funktionen von *TWindow*, um als untergeordnete MDI-Klasse arbeiten zu können.

Üblicherweise leiten Sie neue Klassen sowohl von *TMDIClient* als auch von *TMDIChild* ab, um den von Ihrer Anwendung verlangten, spezifischen Funktionsumfang bereitzustellen. Die Erzeugung neuer Klassen aus *TMDIClient* und *TMDIChild* zur Unterstützung der Zeichenbrettanwendung wird später in diesem Schritt diskutiert.

Ändern des Rahmenfensters

Der erste Schritt beim Umwandeln der Zeichenbrettanwendung für MDI ist die Änderung des Rahmenfensters. MDI-Anwendungen benutzen spezielle MDI-Rahmenfenster. Wie schon zuvor besprochen, bietet ObjectWindows die beiden MDI-Rahmenfensterklassen *TMDIFrame* und *TDecoratedMDIFrame.*

Der Konstruktor für *TDecoratedMDIFrame* sieht folgendermaßen aus:

```
TDecoratedMDIFrame(const char far* title,
                   TResId menuResId,
                   TMDIClient& clientWnd = *new TMDIClient,
                   bool trackMenuSelection = false,
                   TModule* module = 0);
```

Dabei gilt:

- *title* ist die Überschrift für das Rahmenfenster.
- *menuResId* ist der Ressourcen-Bezeichner für das Hauptmenü des Rahmenfensters.
- *clientWnd* ist das MDI-Client-Fenster für das Rahmenfenster.
- *trackMenuSelection* gibt an, ob der Auswahl der Menüpunkte nachzugehen ist. Dies entspricht der Handhabung des Menüs für den im vorhergehenden Schritt konstruierten *TDecoratedFrame*.
- *module* ist ein Zeiger auf ein Programmodul und wird zur Initialisierung des *Twindow* Basisobjekts verwendet.

Außer dem Hinzufügen der Header-Datei owl\decmdifr.h sind noch zwei weitere Änderungen nötig, um *TDecoratedMDIFrame* in der Beispielanwendung benutzen zu können. Zuerst muß die Zeile, in der das Rahmenfenster erzeugt wird, in der Funktion *TDrawApp::InitMainWindow* verändert werden.

```
TDecoratedMDIFrame *frame = new TDecoratedMDIFrame("Drawing Pad",
                                                   TResId("COMMANDS"),
                                                   *new TDrawMDIClient,
                                                   true);
```

Wie schon zuvor, ist “Drawing Pad” die Überschrift für das Rahmenfenster, welches mit der Menüressource COMMANDS initialisiert wird. Das Client-Fenster ist ein neuer *TDrawMDIClient*, also einer aus *TMDIClient*

abgeleiteten Klasse, die Sie etwas später in diesem Schritt noch definieren werden. Der letzte Parameter gibt an, daß in diesem Fenster die Menünachführung einzuschalten ist. Der *module*-Parameter behält seinen Vorgabewert 0.

In der zweiten Änderung wird der Aufruf von *AssignMenu* am Ende der Funktion *InitMainWindow* aus Schritt 10 gelöscht. Dieser Aufruf ist nicht mehr nötig, da die Menüressource durch den zweiten Parameter des *TDecoratedMDIFrame* Konstruktors festgelegt wird.

Ihre Funktion *InitMainWindow* müßte nun etwa wie folgt aussehen:

```
void
TDrawApp::InitMainWindow()
{
   // Erzeugung eines dekorierten MDI-Rahmens

   TDecoratedMDIFrame *frame = new TDecoratedMDIFrame("Drawing Pad",
TResId("COMMANDS"),
                                                      *new
TDrawMDIClient,
                                                      true);

  // Konstruktion einer Statuszeile
   TStatusBar* sb = new TStatusBar(frame, TGadget::Recessed);

   // Konstruktion einer Dialogelementleiste
   TControlBar *cb = new TControlBar(frame);
   cb->Insert(*new TButtonGadget(CM_FILENEW, CM_FILENEW,
TButtonGadget::Command));
   cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN,
TButtonGadget::Command));
   cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE,
TButtonGadget::Command));
   cb->Insert(*new TButtonGadget(CM_FILESAVEAS, CM_FILESAVEAS,
TButtonGadget::Command));
   cb->Insert(*new TSeparatorGadget);
   cb->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE,
TButtonGadget::Command));
   cb->Insert(*new TButtonGadget(CM_PENCOLOR, CM_PENCOLOR,
TButtonGadget::Command));
   cb->Insert(*new TSeparatorGadget);
   cb->Insert(*new TButtonGadget(CM_ABOUT, CM_ABOUT,
TButtonGadget::Command));

   // Einfügen der Statuszeile und der Dialogelementleiste in den
Rahmen
   frame->Insert(*sb, TDecoratedFrame::Bottom);
   frame->Insert(*sb, TDecoratedFrame::Top);
```

```
    // Einrichten des Hauptfensters und seines Menüs
    SetMainWindow(frame);
}
```

Dieses sind die einzigen in der Klasse *TDrawApp* nötigen Änderungen zur Unterstützung der MDI-Funktionalität.

Erzeugen der MDI-Fensterklassen

Der in der Klasse *TDrawWindow* des vorangegangenen Schrittes enthaltene Funktionsumfang muß im MDI-Modell in zwei Klassen unterteilt werden, weil es zwei Fenster gibt, die Botschaften und Benutzereingaben handhaben:

- Das MDI-Client-Fenster wird während der Konstruktion der MDI-Rahmenklasse erzeugt. Das Fenster bleibt so lange offen wie das Rahmenfenster noch geöffnet ist (in diesem Fall während der Lebenszeit der Anwendung). Dieses Fenster handhabt die Befehle CM_FILEOPEN, CM_FILENEW und CM_ABOUT.
  Beim ersten Starten der Anwendung oder wenn keine Abbildungen geöffnet sind, sind die einzigen sinnvollen Befehle das Öffnen von Bilddateien, das Erzeugen neuer Bilder und das Öffnen des Info-Fensters. Andere in der Beispielanwendung verfügbare Befehle, wie das Speichern von Bildern, die Änderung der Stiftgröße oder der Farbe usw., beziehen sich auf eine bestimmte Abbildung, die bereits geöffnet und in einem untergeordneten Fenster angezeigt sein muß.
- Untergeordnete MDI-Fenster werden als Antwort auf einen der vom Client-Fenster bearbeiteten Befehle CM-FILENEW oder CM_FILEOPEN durch das MDI-Client-Fenster erzeugt. In der Beispielanwendung bearbeiten untergeordnete MDI-Fenster die nicht von *TDrawMDIClient* bearbeiteten Ereignisse, die in Schritt 10 von *TDrawWindow* gehandhabt werden:
  - WM_LBUTTONDOWN
  - WM_RBUTTONDOWN
  - WM_MOUSEMOVE
  - WM_LBUTTONUP
  - CM_FILESAVE
  - CM_FILESAVEAS
  - CM_PENSIZE
  - CM_PENCOLOR

Beachten Sie, daß zu jedem dieser Befehle eine spezifische Abbildung bzw. ein Fenster gehört, d.h. jedes dieser Ereignisse ergibt nur im Zusam-

menhang mit einer in einem untergeordneten Fenster geöffneten Abbildung einen Sinn. So muß beispielsweise bereits eine Abbildung geöffnet sein, wenn der Benutzer der Anwendung eine solche speichern will. Sie sehen also den Unterschied zu den vom MDI-Client-Fenster behandelten Ereignissen, welche entweder ein neues untergeordnetes Fenster mit einer neuen oder bereits existierenden Abbildung darin öffnen oder überhaupt von einer Abbildung unabhängig sind.

Die nächsten Abschnitte behandeln die Erzeugung der MDI-Client-Klasse und der untergeordneten Fensterklasse für die Beispielanwendung.

Erzeugen der untergeordneten MDI-Fensterklasse

Dazu müssen Sie eine Klassendeklaration für die Klasse *TDrawMDIChild* erzeugen und die Funktionen dieser Klasse definieren. Sie können dabei den größten Teil der Klassendeklaration von *TDrawWindow* aus Schritt 10 samt den meisten Funktionen, die nur leicht zu ändern sind, verwenden.

Deklaration der Klasse TDrawMDIChild

Die Klassendeklaration für *TDrawMDIChild* ist sehr ähnlich der Deklaration der Klasse *TDrawWindow* aus Schritt 10. Hier sind die durchzuführenden Änderungen:

- Ändern Sie jedes Auftreten von *TDrawWindow* in *TDrawMDIChild*. Dazu gehört auch der Destruktor, der sich sonst nicht ändert.
- Entfernen Sie die Funktionen *CmFileNew*, *CmFileOpen* und *CmAbout* aus der Klassendeklaration.
- Der Konstruktor für *TMDIChild* verlangt eine *TMDIClient* Referenz anstelle des *TWindow* * aus *TDrawWindow*. Dieser Parameter bezeichnet die übergeordnete Klasse des untergeordneten MDI-Fensters. In diesem Fall wollen Sie dem Konstruktor eine *TDrawMDIClient*-Referenz hinzufügen und ihn dem *TMDIChild* Konstruktor übergeben. Außerdem sollten Sie ein **char*** für die Überschrift des untergeordneten MDI-Fensters anfügen.
- Entfernen Sie aus der Antworttabelle die Einträge für die Handhabung der Ereignisse CM_FILENEW, CM_FILEOPEN und CM_ABOUT.

Ihre Klassendeklaration sollte nun etwa wie folgt aussehen:

```
class TDrawMDIChild : public TMDIChild
{
public:
   TDrawMDIChild(TDrawMDIClient& parent, const char* title = 0);
  ~TDrawMDIChild() { delete DragDC; delete Line; delete Lines; delete
FileData; }

protected:
   TDC *DragDC;
   TPen *Pen;
   TLines *Lines;
```

```
    TLine *Line; // zur Aufnahme einer Zeile, die später in Lines
untergebracht wird
    TOpenSaveDialog::TData
              *FileData;
   bool IsDirty, IsNewFile;

  void GetPenSize(); // GetPenSize ermöglicht Aufrufe der Form
Line->SetPen()

    // Überschreiben der Elementfunktion von TWindow
    bool CanClose();

    // Botschaftsantwortfunktionen
    void EvLButtonDown(uint, TPoint&);
    void EvRButtonDown(uint, TPoint&);
    void EvMouseMove(uint, TPoint&);
    void EvLButtonUp(uint, TPoint&);
    void Paint(TDC&, bool, TRect&);
    void CmFileSave();
    void CmFileSaveAs();
    void CmPenSize();
    void CmPenColor();
    void SaveFile();
    void OpenFile();

    DECLARE_RESPONSE_TABLE(TDrawMDIChild);
};

DEFINE_RESPONSE_TABLE1(TDrawMDIChild, TWindow)
    EV_WM_LBUTTONDOWN
    EV_WM_RBUTTONDOWN
    EV_WM_MOUSEMOVE
    EV_WM_LBUTTONUP
    EV_COMMAND(CM_FILESAVE, CmFileSave),
    EV_COMMAND(CM_FILESAVEAS, CmFileSaveAs),
    EV_COMMAND(CM_PENSIZE, CmPenSize),
    EV_COMMAND(CM_PENCOLOR, CmPenColor),
END_RESPONSE_TABLE;
```

Erzeugen der Funktionen für TDrawMDIChild

Fast alles zu den Funktionen in *TDrawMDIChild* kann aus der Klasse *TDrawWindow* übernommen werden. Sie müssen lediglich die Klassenbezeichner in den Funktionsdeklarationen von *TDrawWindow* in *TDrawMDIChild* umändern. So ändert sich beispielsweise die Deklaration für *EvLButtonDown* von

```
void
TDrawWindow::EvLButtonDown(uint, TPoint& point)
{
```

```
    ⋮
}
```

zu

```
void
TDrawMDIChild::EvLButtonDown(uint, TPoint& point)
{
  ⋮
}
```

Ändern Sie die Klassenbezeichner für folgende Funktionen:

GetPenSize	CanClose
EvLButtonDown	EvRButtonDown
EvMouseMove	EvLButtonUp
Paint	CmFileSave
CmFileSaveAs	CmPenSize
CmPenColor	SaveFile
OpenFile	

Außerdem müssen Sie noch eine geringfügige Änderung in der Funktion *CmFileSaveAs* vornehmen. Da sich der Name einer Abbildung gewöhnlich ändert, wenn ein Benutzer den Befehl *Datei / Speichern unter* verwendet, müssen Sie auch die Überschrift des Fensters auf diesen Dateinamen setzen. Zu diesem Zweck ist die Funktion *SetCaption* zu verwenden; diese nimmt ein **char***, was in diesem Fall das Element *FileName* des *FileData*-Objektes sein sollte. Die Funktion *CmFileSaveAs* sollte nun folgendermaßen aussehen:

```
void
TDrawMDIChild::CmFileSaveAs()
{
   if (IsNewFile)
      strcpy(FileData->FileName, "");
   if ((TFileSaveDialog(this, *FileData)).Execute() == IDOK)
      SaveFile();
   SetCaption(FileData->FileName);
}
```

Erzeugen des TDrawMDIChild Konstruktors

Abgesehen davon, daß *TDrawMDIChild* drei Funktionen weniger als *TDrawWindow* besitzt, besteht der Hauptunterschied zwischen den Klassen *TDrawMDIChild* und *TDrawWindow* im Konstruktor.

Initialisieren der Datenelemente

Ebenso wie *TDrawWindow* enthält auch *TDrawMDIChild* das Gerätekontextobjekt, welches die Abbildung anzeigt und die Arrays, die die Informationen zur Liniendarstellung enthalten, verwaltet. Außerdem enthält sie das *IsDirty* Flag und setzt dieses auf falsch, wenn die Abbildung erzeugt oder geöffnet wird, bzw. wahr, wenn die Abbildung geändert wird. Folglich müssen die Werte, der in diesen Funktionen enthaltenden Varia-

blen – *DragDC*, *Line*, *Lines* und *IsDirty* – im Konstruktor von *TDrawMDIChild* initialisiert werden. Diese Initialisierung sieht genauso aus wie die in der Klasse *TDrawWindow*.

```
DragDC = 0;
Lines = new TLines(5, 0, 5);
Line = new TLine(TColor::Black, 1);
IsDirty = false;
```

Allerdings gibt es hier einige beachtenswerte Änderungen gegenüber dem *TDrawWindow* Konstruktor. Erstens wird die Funktion *Init* nicht mehr aufgerufen. *TMDIChild* stellt keine *Init* Funktion zur Verfügung. Statt dessen sollten Sie einfach den Basisklassenkonstruktor in der Initialisierungsliste von *TDrawMDIChild* aufrufen:

```
TDrawMDIChild::TDrawMDIChild(TDrawMDIClient& parent, const char* title)
   : TMDIChild(parent, title)
   {
   .
   .
   .
}
```

Initialisieren der Datenelemente für die Dateiinformation

Sie können nicht länger einfach annehmen, daß Sie mit jedem Öffnen eines neuen Fensters eine neue Abbildung erzeugen, und somit einfach die Variable *IsNewFile* mit wahr initialisieren. In den vorangegangenen Schritten galt diese Annahme: Bei der Erzeugung eines Fensters war noch keine Datei geöffnet, aber das Fenster war bereit, um darin zu zeichnen. Das *IsNewFile* Flag wurde nur dann auf falsch gesetzt, wenn eine Abbildung entweder in eine Datei abgespeichert oder eine existierende Abbildung aus einer Datei in ein bereits erzeugtes Fenster geholt wurde.

In diesem Fall übernimmt das übergeordnete MDI-Client-Fenster das Erzeugen und Öffnen der Datei. Dann erzeugt es ein untergeordnetes Fenster, das die neue oder existierende Abbildung beinhalten wird. Das untergeordnete Fenster hat vom übergeordneten Fenster herauszubekommen, ob es sich um eine neue Abbildung oder eine existierende aus einer Datei handelt.

Aus selbigem Grund erzeugt das untergeordnete MDI-Fenster nicht unbedingt das *TOpenSaveDialog::TData* Objekt, auf das das *FileData* Element verweist. Die Klasse *TDrawMDIClient* besitzt eine Funktion (oder wird eine solche besitzen, wenn Sie sie erst erzeugt haben) namens *GetFileData*. Diese Funktion hat keine Parameter und gibt einen Zeiger an ein *TOpenSaveDialog::TData* Objekt zurück. Erzeugt das MDI-Client-Fenster das untergeordnete Fenster als Antwort auf ein CM_FILEOPEN Ereignis, so erzeugt es ein neues, die Informationen über die zu öffnende Datei enthaltendes *TOpenSaveDialog::TData* Objekt. *GetFileData* gibt einen Zeiger auf dieses Objekt zurück. Falls aber das MDI-Client-Fenster das untergeordnete Fenster als Antwort auf ein CM_FILENEW Ereignis erzeugt, so

erzeugt *TDrawMDIClient* kein *TOpenSaveDialog::TData* Objekt und *GetFileData* gibt 0 zurück.

Also kann das untergeordnete MDI-Fenster durch Überprüfung des Rückgabewert s von *GetFileData* herausfinden, ob es sich um eine neue Abbildung handelt oder nicht. Gibt *GetFileData* ein gültiges Objekt zurück, dann sollte das untergeordnete Fenster den Zeiger auf dieses Objekt seinem *FileData* Element zuweisen und *IsNewFile* auf falsch setzen. Danach kann es die Funktion *OpenFile* aufrufen, um wie gewöhnlich die Abbildung zu laden. Gibt aber *GetFileData* kein gültiges Objekt zurück (also, mit anderen Worten, den Wert 0), dann sollte das untergeordnete MDI-Fenster *IsNewFile* auf wahr setzen und ein neues *TOpenSaveDialog::TData* Objekt erzeugen. Der Dateiname in diesem neuen Objekt wird, ebenso wie in den bisherigen Schritten, von der *CmFileSaveAs* Funktion gesetzt.

Der Konstruktor von *TDrawMDIChild* sollte wie folgt aussehen:

```
TDrawMDIChild::TDrawMDIChild(TDrawMDIClient& parent, const char* title)
   : TMDIChild(parent, title)
{
   DragDC = 0;
   Lines = new TLines(5, 0, 5);
   Line = new TLine(TColor::Black, 1);
   IsDirty = false;

   // Gibt das übergeordnete ein gültiges FileData Element zurück, so
ist es ein Öffnen.
   // Kopieren des FileData Elements vom übergeordneten, bevor es
verloren geht.
   if (FileData = parent.GetFileData()) {
      // keine neue Datei
      IsNewFile = false;
      OpenFile();
   }
   // Gibt aber das übergeordnete 0 zurück, so ist es ein Erzeugen.
   else {
      // eine neue Datei
      IsNewFile = true;
      // Erzeugen eines neuen FileData Elementes
      FileData = new
TOpenSaveDialog::TData(OFN_HIDEREADONLY|OFN_FILEMUSTEXIST,
                                                "Point Files
(*.PTS)|*.pts|",
                                                0, "", "PTS");
   }
}
```

Beachten Sie, daß im Fall des Öffnens einer Datei das untergeordnete Fenster den von *GetFileData* zurückgegebenen Zeiger seinem *FileData* -Ele-

ment zuweist. Ist dies erst geschehen, so übernimmt das untergeordnete Fenster die Verantwortung für das *TOpenSaveDialog::TData* Objekt inklusive dessen Entfernens. Da letzteres schon im Destruktor passiert, brauchen Sie sich nicht weiter darum zu kümmern.

Erzeugen der MDI-Client-Fensterklasse

Die Klasse *TDrawMDIClient* verwaltet die diversen, in seinem Client-Bereich geöffneten untergeordneten Fenster sowie die gesamten helfenden Funktionen, wie das Erzeugen neuer untergeordneter Fenster, das Schließen eines oder aller Fenster, das Anordnen nebeneinander liegender oder überlappender Fenster sowie das Anordnen von Symbolen dargestellten untergeordneter Fenster. *TDrawMDIClient* erbt einen großen Teil seines Funktionsumfanges von der *TMDIClient* Klasse.

Funktionsumfang von TMDIClient

Es ist wichtig, die Klasse *TMDIClient* zu verstehen, und zwar im wesentlichen, weil sie eine ganze Menge Arbeit übernimmt. *TMDIClient* ist virtuell von der Klasse *TWindow* abgeleitet. Sie überschreibt die beiden virtuellen Funktionen *PreProcessMsg* und *Create* von *TWindow*, um eine spezielle Tastatur- und Menühandhabung gemäß den Bedürfnissen des Client-Fensters zu bieten. *TMDIClient* bearbeitet außerdem einige Ereignisse, die in Tabelle 2.2 beschrieben sind.

Tabelle 2.2: Von *TMDIClient* bearbeitete Ereignisse

Ereignis	Antwortfunktion	Zweck
CM_CREATECHILD	*CmCreateChild*	Erzeugt ein neues untergeordnetes MDI-Fenster
CM_TILECHILDREN	*CmTileChildren*	Ordnet alle nicht als Symbol dargestellten untergeordneten Fenster übereinander an
CM_TILECHILDRENHORIZ	*CmTileChildrenHoriz*	Ordnet alle nicht als Symbol dargestellten untergeordneten Fenster nebeneinander an
CM_CASCADECHILDREN	*CmCascadeChildren*	Ordnet alle nicht als Symbol dargestellten untergeordneten Fenster überlappend an
CM_ARRANGEICONS	*CmArrangeIcons*	Ordnet die Symbole aller untergeordneten Fenster an
CM_CLOSECHILDREN	*CmCloseChildren*	Schließt alle geöffneten untergeordneten MDI-Fenster

Die Zeichenbrettanwendung bietet allerdings nur Menüeinträge für vier von diesen: CM_TILECHILDREN, CM_CASCADECHILDREN, CM_ARRANGEICONS und CM_CLOSECHILDREN.

Diese Antwortfunktionen sind lediglich Hüllen für andere *TMDIClient* Funktionen, die die eigentlichen notwendigen Arbeiten ausführen. Eine jede Antwortfunktion ruft eine Funktion mit demselben Namen, aber ohne den Cm-Präfix auf, also etwa *CmCreateChild* die Funktion *CreateChild*. Die einzige Ausnahme bildet *CmTileChildrenHoriz*, die die *Ti-*

leChildren Funktion mit dem Parameter MDITITLE_HORIZONTAL aufruft.

Eine weitere von *TMDIClient* gelieferte Funktion ist *GetActiveMDIChild*, welche einen Zeiger auf das aktive untergeordnete MDI-Fenster zurückgibt. Beachten Sie, daß es zu einem Zeitpunkt immer nur ein aktives untergeordnetes MDI-Fenster geben kann; aber es gibt auch immer eines, selbst wenn alle untergeordneten MDI-Fenster als Symbol dargestellt sind.

Des weiteren ist noch die Funktion *InitChild* zu betrachten. Dies ist die einzige Funktion in *TMDIClient*, die Sie in *TDrawMDIClient* überschreiben müssen. *InitChild* und das Überschreiben dieser Funktion, damit sie mit *TDrawMDIClient* arbeiten kann, wird weiter oben erklärt.

Datenelemente in TDrawMDIClient

TDrawMDIClient braucht zwei neue Datenelemente, die beide als privat deklariert werden sollten.

Das erste ist *NewChildNum*. Die einzige Aufgabe dieser Variablen besteht darin, sich die Nummer der durch die Funktion *CmFileNew* erzeugten neuen Abbildung zu merken. Diese Nummer wird für die Fensterüberschrift von neuen Abbildungen benötigt. Sie wird vom *TDrawMDIClient* Konstruktor mit 0 initialisiert.

Das zweite ist *FileData*, ein Zeiger auf ein *TOpenSaveDialog::TData* Objekt, also wie das *FileData* Element von *TDrawMDIChild*. *FileData* wird zum Aufbewahren der Dateiinformationen einer vom Benutzer geöffneten, existierenden Datei verwendet. Im Konstruktor wird er auf 0 gesetzt und ebenfalls, sobald das untergeordnete MDI-Fenster geöffnet ist. Wie vormals gezeigt, wird das von *GetFileData* zurückgegebene Objekt dem *FileData* Element von *TDrawMDIChild* zugewiesen. Das von *GetFileData* zurückgegebene Objekt ist eigentlich das, auf das das *FileData* Element von *TDrawMDIClient* zeigt, bzw. es gibt keines im Fall einer neuen Datei.

Hinzufügen von Antwortfunktionen

Außer den von *TMDIClient* gehandhabten Ereignissen bearbeitet *TDrawMDIClient* auch solche, die zuvor von *TDrawWindow* gehandhabt wurden, nicht aber von *TDrawMDIChild*: CM_FILENEW, CM_FILEOPEN und CM_ABOUT. Die Antwortfunktion *CmAbout* ist im wesentlichen unverändert die der *TDrawWindow*-Version, abgesehen vom Ändern der Klassenbezeichner. Die Funktionen *CmFileNew* und *CmFileOpen* müssen hingegen grundlegend geändert werden.

CmFileNew

Die Funktion *CmFileNew* vereinfacht sich gegenüber der *TDrawWindow* Version. Sie braucht sich nicht länger um das Leeren der Zeilen-Arrays, das für ungültig Erklären der Fenster und das Setzen der Flags zu kümmern. Statt dessen setzt sie *FileData* auf 0, so daß das untergeordnete MDI-Fenster die Anzeige einer neuen Abbildung verkünden, *NewChild-*

Num erhöhen und dann *CreateChild* aufrufen kann. *CreateChild* ist die Funktion, die dann tatsächlich das neue untergeordnete MDI-Fenster erzeugt und anzeigt. Sie wird später noch genauer im Zusammenhang mit der Funktion *InitChild* betrachtet.

Die Funktion *CmFileNew* müßte nun folgendermaßen aussehen:

```
void
TDrawMDIClient::CmNewFile()
{
   FileData = 0;
   NewChildNum++;
   CreateChild();
}
```

CmFileOpen

Es gibt einige Unterschiede zwischen der *TDrawWindow* Version und der *TDrawMDIClient* Version von *CmFileOpen*.

- Die *TDrawMDIClient* Version braucht nicht mehr die Funktion *CanClose* aufzurufen, da zum Öffnen eines neuen Fensters kein anderes Fenster geschlossen werden muß.
- *TDrawMDIClient* muß ein neues *TOpenSaveDialog::TData* Objekt zur Verwendung mit dem *TFileOpenDialog* Objekt erzeugen.
- Gibt der Aufruf von *TFileOpenDialog.Execute* ID_OK zurück, so ruft die *TDrawMDIClient* Version *CreateChild* an Stelle von *OpenFile* auf.
- Nach Beendigung des Aufrufs von *CreateChild* ist *FileData* auf 0 zu setzen. Es mag zwar so aussehen, als sollten Sie zunächst das *FileData* Objekt löschen, bevor Sie den Zeiger darauf entfernen aber das Objekt wird in Wirklichkeit vom untergeordneten MDI-Objekt übernommen, welches das Objekt auch löscht, wenn das untergeordneten MDI-Fenster entfernt wird.

Ihre *CmFileOpen* Funktion sollte nun wie folgt aussehen:

```
void
TDrawMDIClient::CmFileOpen()
{
   // Erzeugen von FileData
   FileData = new
TOpenSaveDialog::TData(OFN_HIDEREADONLY|OFN_FILEMUSTEXIST,
                                          "Point Files (*.PTS)|*.pts|",
                                          0, "", "PTS");
   // Wenn das Öffnen der Datei klappt, ...
   if ((TFileOpenDialog(this, *FileData)).Execute() ==IDOK)
      // Erzeuge ein untergeordnetes Fenster.
      CreateChild();
   // FileData wird nicht mehr gebraucht.
   FileData = 0;
}
```

GetFileData

Die einzige für *TDrawMDIClient* neu benötigte Funktion ist *GetFileData*. Diese Funktion wird von *TDrawMDIChild* in ihrem Konstruktor aufgerufen, sie hat keine Parameter und gibt einen Zeiger auf ein *TOpenSaveDialog::TData* Objekt zurück. Ihre Aufgabe besteht darin, einen Zeiger auf das Objekt zu liefern, auf das das *FileData* Element von *TDrawMDIClient* zeigt. Wenn *FileData* ein gültiges Objekt bezeichnet (also während einer Öffnen-Operation), dann sollte *GetFileData FileData* zurückgeben. Bezeichnet *FileData* kein gültiges Objekt (also während der Operation für eine neue Datei), so muß *GetFileData* 0 liefern.

Die eigentliche Funktionsdefinition ist sehr einfach und kann durch Funktionsdefinition in der Klassendeklaration eingefügt werden. Ihr *GetFileData* sollte dann wie folgt aussehen:

```
TOpenSaveDialog::TData *GetFileData() { return FileData ? FileData : 0; }
```

Überschreiben von InitChild

Die einzige von *TDrawMDIChild* überschriebene *TMDIClient* Funktion ist *InitChild*. *InitChild* hat keine Parameter und gibt einen Zeiger auf ein *TMDIChild* Objekt zurück. Die Funktion *CreateChild* ruft *InitChild* vor dem Erzeugen eines neuen untergeordneten MDI-Fensters auf. In *InitChild* erzeugen Sie das *TMDIChild* oder von *TMDIChild* abgeleitete Objekt für das untergeordnete MDI-Fenster. *InitChild* ist die einzige Funktion, die Sie beim Erzeugen der Klasse *TDrawMDIClient* überschreiben.

Die Funktionsweise von *InitChild* aus *TDrawMDIClient* ist ziemlich offensichtlich. Ist *FileData* 0, so erzeugen Sie ein Zeichen-Array für die Standardfensterüberschrift, welche unter Verwendung des Wertes von *NewChildNum* initialisiert werden kann, so daß jede neue Abbildung eine andere Überschrift hat.

Dann sollten Sie ein *TMDIChild** und ein neues *TDrawMDIChild* Objekt erzeugen. Der Konstruktor für *TDrawMDIChild* nimmt zwei Parameter entgegen, und zwar einen Verweis auf ein *TDrawMDIClient* Objekt für das übergeordnete Fenster und einen **const char***, der die Überschrift des untergeordneten MDI-Fensters beinhaltet. In diesem Fall sollte der erste Parameter durch den **this**-Zeiger dereferenziert werden. Der zweite Parameter ist entweder (falls *FileData* auf ein gültiges Objekt zeigt) ein *FileName* Element des *FileData* Objekts oder (sonst) das von Ihnen zuvor erzeugte Zeichen-Array.

Sobald das untergeordnete MDI-Objekt erzeugt worden ist, müssen Sie die Funktion *SetIcon* für das Objekt aufrufen. *SetIcon* assoziiert eine Symbolressource mit dem Objekt der Funktion. Dieses Symbol wird angezeigt, wenn das untergeordnete Fenster als Symbol darzustellen ist. Sie können als Symbol etwa das der Beispielanwendung namens IDI_TUTORIAL verwenden.

Als letztes hat die Funktion den *TMDIChild* Zeiger zurückzugeben. Ihre *InitChild* Funktion müßte nun etwa folgendermaßen aussehen:

```
TMDIChild*
TDrawMDIClient::InitChild()
{
   char title[15];
   if (!FileData)
      wsprintf(title, "New drawing %d", NewChildNum);
   TMDIChild* child = new TDrawMDIChild(*this, FileData ?
                                        FileData->FileName : title);
   child->SetIcon(GetApplication(), TResId("IDI_TUTORIAL"));
   return child;
}
```

Schritt 12: Übergang zum Dok/View-Modell

In Schritt 12 erfolgt eine Einführung in das Programmiermodell Dok/View, das auf dem Prinzip der Trennung von Daten und Oberfläche für diese Daten beruht.

Im wesentlichen werden die Daten in einem von der *TDocument*-Klasse abgeleiteten Dokumentobjekt eingekapselt, auf dem Bildschirm ausgegeben und vom Benutzer über ein von der *TView*-Klasse abgeleitetes View-Objekt bearbeitet.

Sie finden den Quelltext zu Schritt 12 in den Dateien STEP11.CPP, STEP11.RC, STEP11DV.CPP und STEP11DV.RC im Verzeichnis EXAMPLES\OWL\TUTORIAL.

Das Dok/View-Modell bietet einen höheren Grad an Flexibilität bei der Darstellung von Daten als ein Modell, das die Dateneinkapselung und die Benutzeroberfläche in einer einzigen Klasse verknüpft. Über das Dok/View-Modell können Sie eine Objektklasse definieren, die alle Datentypen, wie z.B. eine einfache Textdatei, eine Datenbankdatei oder (wie in unserem Beispielprogramm) eine Zeilenausgabe enthält. Danach sind Sie dann in der Lage, viele verschiedene View-Klassen zu erzeugen. Alle diese Klassen zeigen die gleichen Daten in einer anderen Form an oder ermöglichen die interaktive Datenbearbeitung durch den Benutzer auf eine andere Art und Weise.

In Schritt 12 werden Sie jedoch lediglich das derzeit für die Beispielanwendung genutzte Modell in das Dok/View-Modell umwandeln. Der Quelltext zu Schritt 12 besitzt ein völlig anderes Aussehen als der zu Schritt 10, aber die aus Schritt 12 resultierende laufende Anwendung ist fast identisch mit der Anwendung aus Schritt 10.

Organisieren des Quelltextes einer Anwendung

Der Quelltext für Schritt 12 wird in vier Quelltextdateien unterteilt:

- STEP11.CPP enthält das Anwendungsobjekt und seine Elementdefinitionen. Die Datei enthält ebenfalls die Funktion *OwlMain*.
- STEP11.RC enthält die Bezeichner für die Ereignisse, die durch das Anwendungsobjekt gesteuert werden, die Ressourcen für das Rahmenfenster und seine Dekorationen, das Dialogfenster Info und das Anwendungsmenü.
- STEP11DV.CPP enthält die *TLine*-Klasse, die Objektklasse *TDrawObjekt*, die View-Klasse *TDrawView* und die Definitionen der mit diesen Klassen verknüpften Elementfunktionen.
- STEP11DV.RC enthält die Bezeichner für die Ereignisse, die durch das View-Objekt gesteuert werden, sowie die Ressourcen für das View.

Sie sollten Ihren Quelltext ähnlich unterteilen, um so zwischen dem Objekt, dem unterstützenden View und dem Quelltext der Anwendung unterscheiden zu können. Der Anwendungs-Quelltext enthält die Struktur, in die das Objekt und die View-Klassen eingebunden werden, nicht aber die Funktionalität des Dok/View-Modells selbst. Dadurch zeichnet sich eine Entwicklungsmethode aus, mit der Quelltext geschrieben wird, der beliebig wiederverwendet werden kann.

Das Dok/View-Modell

Das Dok/View-Modell beruht auf drei ObjektWindows-Klassen:

- Die KLasse *TDocument* enthält verschiedene Daten und steuert den Zugriff auf diese Daten. Ein Dokumentobjekt organisiert den Zugriff des Benutzers auf diese Daten, indem die Eingabe über verknüpfte View-Objekte durchgeführt wird. Es ist möglich, ein Dokumentobjekt mit mehreren Views gleichzeitig zu verknüpfen (um der Einfachheit unseres Beispiels willen wird das Dokumentobjekt jedoch nur mit einem einzigen View-Objekt verbunden).
- Die Klasse *TView l*iefert die Schnittstelle zwischen einem Dokumentobjekt und der Benutzeroberfläche. Über ein View-Objekt wird gesteuert, wie die Daten aus dem Dokumentobjekt auf dem Bildschirm angezeigt werden. Ein View-Objekt kann immer nur mit einem Dokumentobjekt gleichzeitig verbunden sein.
- Die Klasse *TDocManager* koordiniert die Verbindungen zwischen einem Dokumentobjekt und seinen View-Objekten. Der Dokumentmanager enthält ein Standardmenü Datei und Standardereignisroutinen für jeden Menüpunkt des Datei-Menüs. Weiterhin enthält diese Klasse eine Liste der Objekt-Templates. Jedes dieser Templates spezifiziert eine bestimmte Beziehung zwischen einer Dokumentklasse und einer View-Klasse.

Die Klassen *TDocument* und *TView* enthalten die abstrakte Funktionalität für die Dokument- und View-Objekte. Die speziellen Funktionen Ihrer eigenen Dokument- und View-Klassen müssen Sie natürlich selbst schreiben. Außerdem muß der Dokumentmanager explizit erzeugt und in das Anwendungsobjekt eingearbeitet werden. Schließlich müssen Sie dem Dokumentmanager auch die Dokument-Templates zur Verfügung stellen. All diese Schritte werden in den folgenden Abschnitten eingehend beschrieben.

Die Klasse TDrawDocument

Die Klasse *TDrawDocument* wurde von der ObjektWindows-Klasse *TFileDocument* abgeleitet, die wiederum von der Klasse *TDocument* abgeleitet wurde. *TDocument* enthält verschiedene Eingabe- und Ausgabefunktionen. Diese **virtuellen** Funktionen liefern Dummy-Werte zurück und besitzen keine wirkliche Funktionalität. *TFileDocument* enthält die Basisfunktionen, die für den Zugriff auf eine Datendatei über einen Stream erforderlich sind.

TDrawDocument verwendet die in *TFileDocument* enthaltenen Funktionen zum Zugriff auf die in einer Datei abgespeicherten Zeilen. Die bereits in den vorhergehenden Schritten verwendeten Zeilen sind im Array *TLines* (dem gleichen wie in vorherhgehenden Schritten) enthalten. Auf das Array wird über den Zeiger *Lines* verwiesen.

Erzeugen und Entfernen einer Instanz von TDrawDocument

Der Konstruktor der Klasse *TDrawDocument* erwartet einen einzigen Parameter: einen *TDocument* ***-Wert. Dieser Wert enthält einen Zeiger auf das übergeordnete Dokument. Ein Dokument kann mehreren anderen Dokumenten übergeordnet sein. Dadurch werden die in den untergeordneten Dokumenten enthaltenen Daten so behandelt, als wären sie ein Teil des übergeordneten Dokuments. Der Konstruktor übergibt den Zeiger auf das übergeordnete Dokument an *TFileDocument*. Er setzt weitherin die in *Lines* enthaltenen Datenelemente zu Beginn auf den Wert 0.

Der Destruktor der Klasse *TDrawDocument* löscht das *TLines*-Dokument, auf das über *Lines* verwiesen wird.

Speichern von Zeilen

Die Dokumentklasse, mit der Sie Dialogelemente erzeugen, greift auf die in einer Darstellung enthaltenen Daten zu. Sie müssen diese Daten aber noch irgendwie abspeichern.

In den vorhergehenden Schritten wurden bereits die Klassen *TLine* und das Array *TLines* erzeugt. Glücklicherweise kann dieser Quelltext wiederverwendet werden. Die Zeilen werden für jedes Objekt in einem *TLines*-Array abgespeichert. Das Objekt greift über das **protected** *TLines* ***-Datenelement mit der Bezeichnung *Lines* auf diese Daten zu.

Die Arrays *TPoints* und *TLines* und deren Iteratoren sowie die Klasse *TLine* werden jetzt in der Datei STEP11DV.CPP definiert. Im Dok/View-Mo-

dell sind diese Klassen ein integrierter Bestandteil der Objektklasse, die Sie gerade erzeugen. Der Quellcode für diese Klassen hat sich seit dem Schritt 10 nicht geändert.

Implementieren virtueller TDocument-Funktionen

In *TDrawDocument* müssen einige virtuelle Funktionen implementiert werden, die von *TDocument* vererbt wurden. Diese Funktionen ermöglichen das Arbeiten mit Streams. Außerdem können über diese Funktionen die seit der letzten Abspeicherung in ein Dokument eingearbeiteten Veränderungen auf der Festplatte abgespeichert oder verworfen werden.

Öffnen und Schließen einer Darstellung

TFileDocument enthält zwar die für die Stream-Ein- und Ausgabe erforderlichen Grundfunktionen, aber keine Funktion zum Lesen der Daten einer Zeile. Wollen Sie diese Möglichkeit einbauen, dann müssen Sie die Funktionen *Open* und *Close* überschreiben, d.h. modifizieren.

Die Syntax der Funktion *Open* sieht folgendermaßen aus:

```
BOOL Open(int mode, const char far* path=0);
```

Dabei ist:

- *mode* der Dateiöffnungsmodus. In unserem Beispiel können Sie den Parameter mode ignorieren; die Datei wird wie jedesmal mit dem Flag *ofRead* geöffnet.
- path enthält die Angabe des Dokument-Verzeichnisses. Wurde ein Pfad angegeben, dann wird der derzeit aktuelle Pfad für das Dokument durch das angegebene Verzeichnis ersetzt. Wird kein Pfad angegeben (das heißt *path* nimmt den vorgegebenen Wert an), dann wird die Pfadangabe nicht verändert. Der Pfad wird vom Dokument für das Erzeugen von Dokument-Streams verwendet.

Die Funktion *Open* ähnelt der Funktion *OpenFile*, die bereits in früheren Einführungschritten verwendet wurde. Zwischen beiden Funktionen gibt es folgende Unterschiede:

- Die Funktion *Open* erzeugt das *TLines*-Array für das Dokumentobjekt. Bisher wurde dies vom *TMyWindow*-Konstruktor durchgeführt, weil *TMyWindow* für das Aufbewahren aller *TLine*-Objekte verantwortlich war. Nun ist das Dokument für diese Aufgabe verantwortlich. Deshalb muß im Dokument Speicherplatz für die Daten bereitgestellt werden, bevor die Daten eingelesen werden.

Wurde ein Wert für *path* übergeben, dann setzt *Open* über die Funktion *SetDocPath* den Pfad des Dokuments auf den angegebenen Pfad.
Open überprüft, ob das Dokument einen Pfad besitzt. Wenn nicht, dann handelt es sich um ein neues Dokument. In diesem Fall ist nicht erforderlich, Daten aus einer Datei einzulesen. Wenn ja, dann ruft *Open* die Funk-

tion *InStream* auf. Diese Funktion ist in *TFileDocument* definiert und gibt einen *TInStream* **_*** zurück.

TInStream ist die Standardklasse für Eingabe-Streams und wird von den Dok/View-Klassen genutzt. *TInStream* wurde von den Klassen *TStream* und *istream* abgeleitet. *TStream* ist eine abstrakte Basisklasse, die den Zugriff von Dokumenten auf Standard-Streams ermöglicht.

TInStream ist im wesentlichen ein Standard-*istream*, der zur Verwendung mit Dok/View-Modell angepaßt wurde. Es gibt auch eine entsprechende *TOutStream*-Klasse, die von *TStream* und *ostream* abgeleitet wurde. Sie sollten die Klasse *TOutStream* zum Erzeugen der Funktion *Commit* verwenden.

- Nachdem der Eingabe-Stream erzeugt wurde, werden die Daten eingelesen und im Array *TLines* abgelegt. *Lines* verweist auf dieses Array. Nach dem Einlesen wird der Eingabe-Stream wieder gelöscht.
- Danach ruft *Open* die Funktion *SetDirty* auf und übergibt den Wert FALSE als Funktionsparameter. *SetDirty* und ihre entsprechende Zugriffsfunktion *isDirty* stellen die Äquivalente zu dem Flag IsDirty in früheren Schritten der vorliegenden Einführung dar. Ein Dokument wird als "dirty" betrachtet, wenn es irgendwelche Datenänderungen enthält, die noch nicht abgespeichert wurden.
- Abschließend liefert die Funktion *Open* einen Rückgabewert. Wurde das Dokument erfolgreich geöffnet, dann ist der Rückgabewert TRUE.

Es folgt ein Programmvorschlag zum Aufbau der *Open*-Funktion:

```
BOOL TDrawDokument::Open(int /*mode*/, const char far* path)
{
  Lines = new TLines(5, 0, 5);
  if (path)
    SetDocPath(path);
  if (GetDocPath()) {
    TInStream* is = InStream(ofRead);
    if (!is)
      return FALSE;

    unsigned numLines;
    char fileinfo[100];
    *is >> numLines;
    is->getline(fileinfo, sizeof(fileinfo));
    while (numLines--) {
      TLine line;
      *is >> line;
      Lines->Add(line);
    }
    delete is;
  }
  SetDirty(FALSE);
  NotifyViews(vnRevert, FALSE);
```

```
  return TRUE;
}
```

Das Schließen einer Darstellung gestaltet sich weniger kompliziert. Die Funktion *Close* verwirft die Daten des Dokuments und führt die Abschlußarbeiten durch. Im vorliegenden Fall (Beispiel) löscht diese Funktion das Array *TLines*, auf das über das Datenelement *Lines* verwiesen wird, und liefert den Rückgabewert TRUE. Es folgt ein Quelltextvorschlag zu der Funktion *Close*:

```
BOOL TDrawDokument::Close()
{
  delete Lines;
  Lines = 0;
  return TRUE;
}
```

Lines wird sowohl über den Konstruktor als auch nach dem Schließen des Dokuments auf 0 gesetzt. Ist das Dokument offen, dann verweist *Lines* auf ein Array mit der Bezeichnung *TLines* und ist daher ungleich 0. Dadurch, daß *Lines* auf 0 gesetzt wird, kann man sehr leicht überprüfen, ob das Dokument offen ist. Über die Funktion *IsOpen* können Sie diesen Umstand auch außerhalb des Dokumentobjekts prüfen:

```
BOOL IsOpen() { return Lines != 0; }
```

Speichern und Verwerfen von Änderungen

TDocument enthält zwei Funktionen zum Speichern und Verwerfen der Änderungen in einem Dokument:

- Die Funktion *Commit* schreibt die über (mit einem Dokument verbundenen) Views eingearbeiteten Änderungen auf die Festplatte und speichert die Daten danach dauerhaft. *Commit* erwartet einen boooleschen Parameter. Ist dieser Parameter FALSE, dann soeichert *Commit* die Daten nur, wenn das Dokument "dirty" ist. Ist der Parameter TRUE, dann führt *Commit* eine vollständige physische Speicherung der Daten durch. Standardwert für diesen Parameter ist FALSE.
- Die Funktion *Revert* verwirft alle Änderungen in den Views des Dokuments. Danach erzwingt die Funktion, daß die Views die im Dokument enthaltenen Daten erneut laden und anzeigen. *Revert* erwartet einen booleschen Parameter. Ist dieser Parameter TRUE, dann löscht das View sein Fenster und lädt die Daten nicht aus dem Dokument nicht nach. Der Standardwert für diesen Parameter ist FALSE.

In der Klasse *TDrawDocument* wird das Dokument aktualisiert, wenn die einzelnen Zeilen in dem View-Fenster ausgegeben werden. Die *Commit*-Funktion für diese Klasse speichert die Daten in einer Datei.

Commit überprüft, ob das Dokument "dirty" ist. Wenn das nicht der Fall ist und wenn der Parameter *force* FALSE ist, dann liefert *Commit* TRUE zu-

rück, wodurch angezeigt wird, daß die Operation erfolgreich abgeschlossen wurde.

Ist das Dokument "dirty" oder ist der Parameter *force* TRUE, dann speichert *Commit* die Daten. Die Prozedur zum Speichern der Daten ähnelt der Funktion *SaveFile* aus zuvor behandelten Schritten. Es gibt aber wie bei der Funktion *Open* einige kleine Unterschiede.

Commit ruft die Funktion *OutStream* auf, damit ein Ausgabe-Stream geöffnet wird. Diese Funktion ist in der Klasse *TFileDocument* definiert und liefert einen *TOutStream* **_*** zurück. Danach schreibt *Commit* die Daten in den Ausgabe-Stream. Diese Prozedur entspricht fast genau der, die für die alte Funktion *SaveFile* verwendet wurde.

Nachdem die Daten in den Ausgabe-Stream geschrieben wurden, schaltet *Commit* das Flag *IsDirty* über den Aufruf der Funktion *SetDirty* mit dem Parameter FALSE aus. Danach gibt *Commit* TRUE zurück, wodurch angezeigt wird, daß die Operation erfolgreich abgeschlossen wurde.

Es folgt ein Programmbeispiel als Vorschlag, wie Ihre *Commit*-Funktion möglicherweise aussehen könnte:

```
BOOL TDrawDokument::Commit(BOOL force)
{
  if (!IsDirty() && !force)
    return TRUE;

  TOutStream* os = OutStream(ofWrite);
  if (!os)
    return FALSE;

  // Ausgabe mehrerer Zeilen
  *os << Lines->GetItemsInContainer();

  // Anhängen einer Beschreibung über einen Ressource-String
  *os << ' ' << string(*GetDocManager().GetApplication(),IDS_FILEINFO)
<< '\n';

  // Angabe eines Iterators für das Zeilenarray
  TLinesIterator i(*Lines);

  // Solange der Iterator gültig ist (d.h., solange noch Zeilen
ausgegeben werden müssen)
  while (i) {
    // Kopiert die aktuellen Zeile vom Iterator und erhöht den
Array-Index um Eins.
    *os << i++;
  }
  delete os;

  SetDirty(FALSE);
  return TRUE;
}
```

In der Funktion *Commit* gibt es nur eine Neuigkeit:

```
// Anhängen einer Beschreibung über einen Ressource-String
  *os << ' ' << string(*GetDocManager().GetApplication(),IDS_FILEINFO)
<< '\n';
```

Hier wird ein spezieller Konstruktor für die ANSI-*string*-Klasse verwendet:

```
string(HINSTANCE instance, UINT id, int len = 255);
```

Über diesen Konstruktor sind Sie in der Lage, sich eine Stringressource aus einer beliebigen Windows-Anwendung heraus ausgeben lassen. Sie spezifizieren die Anwendung durch Übergabe einer HINSTANCE als dem ersten Parameter des *string*.Konstruktors. In unserem Fall können Sie sich die Instanz der aktuellen Anwendung über den Dokumentmanager ausgeben lassen. Die Funktion *GetDocManager* liefert einen Zeiger auf den Dokumentmanager des Dokuments zurück. Umgekehrt liefert die Funktion *GetApplication* einen Zeiger auf die Anwendung, in der der Dokumentmanager enthalten ist. Dieser Zeiger über den Konvertierungsoperator der *TModule*-Klasse implizit in eine HINSTANCE umgewandelt. Der zweite Parameter des *string*-Konstruktors stellt den Ressourcenbezeichner eines String dar, der in STEP11DV.RC definiert ist. Dieser String enthält Versionsinformationen. Diese können zur Identifizierung der Anwendung verwendet werden, in der das Dokument erzeugt wurde.

Die Funktion *Revert* erwartet einen booleschen Parameter. Über diesen wird angezeigt, ob die Bildschirmdarstellung der Dokument-Views mit den Daten des Dokuments neu aufgebaut werden muß. *Revert* ruft die *TFileDocument*-Version der Funktion *Revert* auf, die wiederum die *TDocument*-Version von *Revert* aufruft. Die Basisklassenfunktion aktiviert die Funktion *NotifyViews* mit dem Ereignis *vnRevert*. Der zweite Parameter der Funktion *NotifyViews* wird auf den Parameter gesetzt, der an die Funktion *TDrawDocument::Revert übergeben wurde*.

TFileDocument::Revert setzt *IsDirty* auf FALSE und gibt einen Wert zurück. Wenn *TFileDocument::Revert* FALSE zurückgibt, dann muß *TDrawDocument* ebenfalls FALSE zurückgegeben.

Wenn *TFileDocument::Revert* TRUE zurückgibt, dann muß die Funktion *TDrawDocument* den an *Revert* übergebenen Parameter überprüfen. Ist dieser FALSE (das heißt, wenn die Bildschirmausgabe des View aktualisert werden muß), dann ruft *Revert* die Funktion *Open* auf, um die Dokumentdatei zu öffnen und die Daten erneut zu laden und anzuzeigen.

Betrachten Sie folgenden Programmvorschlag für die Funktion *Commit* an:

```
BOOL TDrawDokument::Revert(BOOL clear)
{
  if (!TFileDokument::Revert(clear))
    return FALSE;
  if (!clear)
    Open(0);
  return TRUE;
}
```

Zugriff auf die Daten eines Dokuments

Es gibt im wesentlichen zwei Möglichkeiten für den Zugriff auf die Daten in *TDrawDocument*: über das Hinzufügen einer Zeile (wenn der Benutzer zum Beispiel eine neue Zeile in ein View eingibt) und über die Ausgabe einer Referenz auf eine Zeile in einem Dokument (z.B. über die Ausgabe einer Referenz auf jede Zeile, wenn das Fenster auf dem Bildschirm erneut ausgegeben wird). Sie können die beiden Funktionen *AddLine* und *GetLine* einarbeiten, um folgende Aktionen abzusichern:

Die Funktion *AddLine* fügt eine neue Zeile in das Array *TLines des* Dokuments ein. Diese Zeile wird als *TLine* **&** an die Funktion *AddLines* übergeben. Nachdem die Zeile in das Array eingefügt wurde, setzt *AddLine* das Flag *IsDirty* auf TRUE, indem sie die Funktion *SetDirty* aufruft. Danach liefert sie die Indexnummer der gerade eingefügten Zeile zurück. Folgendermaßen müßte der Quelltext zu der Funktion *AddLines* gestaltet werden:

```
int TDrawDokument::AddLine(TLine& line)
{
  int index = Lines->GetItemsInContainer();
  Lines->Add(line);
  SetDirty(TRUE);
  return index;
}
```

An die Funktion *GetLine* wird ein **int**-Parameter übergeben. Dieses **int** enthält den Index der entsprechenden Zeile. *GetLine* muß zuerst überprüfen, ob das Dokument offen ist. Wenn nicht, dann kann *GetLine* versuchen, das Dokument zu öffnen. Ist das Dokument nicht offen und läßt es sich auch nicht durch *GetLine* öffnen, dann gibt die Funktion 0 zurück. Dies bedeutet, daß die Funktion kein gültiges Dokument finden konnte, aus der sie eine Zeile entnehmen konnte.

Haben Sie herausgefunden, daß das Dokument gültig ist, dann sollten Sie auch noch überprüfen, ob der Index nicht zu groß ist. Vergleichen Sie den Index mit dem Rückgabewert der Funktion *GetItemsInContainer*. So lange der Index kleiner ist, können Sie einen Zeiger auf das *TLine*-Objekt zurückgeben. Nun folgt ein Codebeispiel dafür, wie Sie Ihre *GetLine*-Funktion schreiben könnten:

```
TLine* TDrawDokument::GetLine(int index)
{
  if (!IsOpen() && !Open(ofRead | ofWrite))
    return 0;
  return index < Lines->GetItemsInContainer() ?
(*Lines)[index] : 0;
}
```

Die Klasse TDrawView

Die Klasse *TDrawView* wurde von der ObjektWindows-Klasse *TWindowView* abgleitet, die wiederum von den Klassen *TView* und *TWindow* abgeleitet wurde. *TView* besitzt keine vererbten Fensterfunktionen. Eine von *TView* abgeleitete Klasse erhält diese Fähigkeiten entweder durch Hinzufügen eines Fensterelements oder Zeigers oder durch das Mischen einer Fensterklasse mit einer View-Klasse.

TWindowView verwendet die letztere Methode und mischt *TWindow* und *TView,* wobei eine Klasse mit grundlegenden Fenster-und View-Funktionen entsteht. Durch Ableitungen von dieser Allzweck-Klasse muß *TDrawView* nur noch die für die Arbeit mit der Klasse *TDrawDocument* erforderlichen Funktionen hinzufügen.

TDrawView ähnelt der Klasse *TMyWindow,* die in den vorhergehenden Schritten verwendet wurde. Sie werden sehen, daß wirklich viele Funktionen mit kleinen oder gar keinen Veränderungen direkt von *TMyWindow* zu *TDrawView* übernommen wurden.

TDrawView-Datenelemente

Die Klasse *TDrawView* besitzt verschiedene **protected**-Datenelemente.

```
TDC *DragDC;
TPen *Pen;
TLine *Line;
TDragDokument *DrawDoc;
```

Drei davon müßten Ihnen bekannt vorkommen. *DragDC, Pen* und *Line* beisitzen in *TDrawView* die gleiche Funktionalität wie in *TMyWindow*.

Ein Dokument kann ohne Views existieren. Die Umkehrung gilt jedoch nicht. Ein View muß immer mit einem existierenden Dokument verknüpft sein.

TDrawView wird an das zugehörige Dokument angepaßt, wenn dieses erzeugt wird. Die Klasse überwacht dieses Dokument über ein *TDrawDocument* **_*,** das *DrawDoc* genannt wird. Die Basisklasse *TView* besitzt ein Element mit der Bezeichnung Doc als *TDocument* **_*,** das die gleichen grundlegenden Aufgaben erfüllt. Während der Erzeugung der Basisklasse wird *Doc* auf das *TDrawDocument*-Objekt gerichtet, das an den *TDrawView*-Konstruktor übergeben wurde. *DrawDoc* wird eingefügt, um Typübereinstimmung zu erzwingen, wenn auf den Dokumentzeiger zugegriffen wird.

Erzeugen einer Instanz der TDrawView-Klasse

Der Konstruktor der Klasse *TDrawView* erwartet zwei Parameter: ein *TDrawDocument* **&** (eine Referenz auf das mit den Views verbundene Dokument) und ein *TWindow* ***** (ein Zeiger auf das übergeordnete Fenster). Der Wert für das übergeordnete Fenster ist standardmäßig 0, wenn kein anderer Wert übergeben wurde. Der Konstruktor übergibt seine zwei Parameter an den *TWindowView*-Konstruktor. Er initialisiert das Element *DrawDoc* so, daß es auf das als erster Parameter übergebenes Dokument zeigt.

Der Konstruktor setzt auch *DragDC* auf 0 und initialisiert *Line* mit einem neuen *TLine*-Objekt.

Abschließend erstellt der Konstruktor das Menü des View. Um einen Menü-Deskriptor aus einer Menüresssource heraus zu erzeugen, können Sie die Klasse *TMenuDescr* verwenden. Es folgt die Syntax des *TMenuDescr*-Konstruktors:

```
TMenuDescr(TResId id, int fg, int eg, int cg, int og, int wg, int hg);
```

Dabei ist:

- *id* der Ressourcenbezeichner der Menüressource.
- fg die Anzahl der Menügruppen in dem Menü Datei.
- eg die Anzahl der Menügruppen in dem Menü Bearbeiten.
- cg die Anzahl der Menügruppen in dem Menü Container.
- og die Anzahl der Menügruppen in dem Menü Objekt.
- wg die Anzahl der Menügruppen in dem Menü Fenster.
- hg die Anzahl der Menügruppen in dem Menü Hilfe.

Obwohl die Gruppen unterschiedliche Namen besitzen, repräsentieren diese Namen nur den allgemeinen Namen der jeweiligen Menügruppe. Das von jeder Gruppe dargestellte Menü muß nicht unbedingt diesen Namen tragen. Der Dokumentmanager enthält ein Standardmenü Datei, die anderen Menübezeichnungen können jedoch in der Menüressource festgelegt werden.

Wenn ein Parameter einer Menügruppe 0 ist, dann deutet das darauf hin, daß die Menüressource kein Menü für diese Gruppe enthält. Die über die Menügruppenparameter angegebene Gesamtanzahl der Menügruppen muß kleiner gleich der Anzahl der Menügruppen sein, die in der Menüressource zur Verfügung stehen.

In unsrem Beispielprogramm liefert das View eine Menüressource mit der Bezeichnung IDM_DRAWVIEW, die in der Datei STEP11DV.RC abgespeichert ist. Dieses Menü wird Tools genannt und besitzt die gleichen Auswahlpunkte wie das Menü Tools in den vorhergehenden Schritten:

Stiftgröße und Stiftfarbe. Um das Menü Tools als zweites Menü in die Menüleiste einzufügen, müssen der Parameter *eg* (der zweite Menügruppenparameter) gleich 1 und die restlichen Menügruppenparameter gleich 0 sein.

Sie können den Menü-Deskriptor als View-Menü TView installieren, indem Sie die *TView*-Funktion *SetViewMenu* mit dem Parameter *TMenuDescr* **_*** aufrufen._SetViewMenu aktiviert den Menü-Deskriptor als View-Menü. Während der Erzeugung des View verschmilzt dieses Menü mit dem Anwendungsmenü.

Es folgt die Syntax des Funktionaufrufs zur Einrichtung des View-Menüs:

```
SetViewMenu(new TMenuDescr(IDM_DRAWVIEW,0,1,0,0,0,0));
```

Der Destruktor für das View löscht den Gerätekontext, auf den über *DragDC* verwiesen wird, sowie das *TLine*-Objekt, auf das über *Line* verwiesen wird.

Benennen der Klasse

Jede View-Klasse muß eine Definition der Funktion *StaticName* enthalten. Diese übernimmt keine Parameter und liefert einen **static const char far *** zurück. In diesem Rückgabewert ist der Namen der View-Klasse enthalten. Es folgt ein mögliche Variante der *StaticName*-Funktion:

```
static const char far* StaticName() {return "Draw View";}
```

Protected-Funktionen

TDrawView besitzt verschiedene **protected**-Zugriffsfunktionen, die die Funktionalität dieser Klasse liefern.

Die Funktion *GetPenSize* ist mit der Funktion *TMyWindow* identisch. Diese Funktion öffnet einen_*TInputDialog*, nimmt vom Benutzer die Angabe der gewünschten Stiftgröße entgegen und verändert die StiftgGröße für das Fenster. Danach ruft sie die Funktion *SetPen* für die aktuelle Zeile auf.

Die *Paint*-Funktion unterscheidet sich ein wenig von der *Paint*-Funktion aus der *TMyWindow*-Klasse, löst aber grundsätzlich dieselben Aufgaben. Statt eines Iterators ruft *TDrawView::Paint* die Funktion *GetLine des* (mit dem View verknüpften) Dokuments zum Bearbeiten der Zeilen in einem Array auf. Der Rückgabewert der Funktion *GetLine* wird einem **const** TLine * mit der Bezeichnung *line* zugewiesen. Ist *line* ungleich 0 (das heißt, wenn *GetLine* eine gültige Zeile zurückgibt), dann ruft *Paint* die Zeilenfunktion *Draw* auf. Die *TLine*-Klasse hat sich seit Schritt 10 nicht mehr verändert. Die Zeile gibt sich selbst im Fenster aus.

Es folgt ein Programmvorschlag für die Funktion *Paint*:

```
void TDrawView::Paint(TDC& dc, BOOL, TRect&)
{
  // Zählt die Zeilenobjekte im Array.
  int i = 0;
  const TLine* line;
  while ((line = DrawDoc->GetLine(i++)) != 0)
    line->Draw(dc);
}
```

Ereignisbehandlung in TDrawView

Die Klasse *TDrawView* behandelt viele Ereignisse, die zuvor von der Klasse *TMyWindow* betreut wurden. Die meisten ursprünglich von *TMyWindow* behandelten anderen Ereignisse werden vom Anwendungsobjekt und vom Dokumentmanager behandelt; dies wird in Schritt 12 zu einem späteren Zeitpunkt beschrieben.

TDrawView reagiert außerdem auf zwei neue Botschaften: VN_COMMIT und VN_REVERT. Diese View-Botschaften werden vom Dokument eines View gesendet, wenn die Dokument-Funktionen *Commit* und *Revert* aufgerufen werden.

Sehen Sie sich die Definition der Antworttabelle für *TDrawView* an:

```
DEFINE_RESPONSE_TABLE1(TDrawView, TWindowView)
  EV_WM_LBUTTONDOWN,
  EV_WM_RBUTTONDOWN,
  EV_WM_MOUSEMOVE,
  EV_WM_LBUTTONUP,
  EV_COMMAND(CM_PENSIZE, CmPenSize),
  EV_COMMAND(CM_PENCOLOR, CmPenColor),
  EV_VN_COMMIT,
  EV_VN_REVERT,
END_RESPONSE_TABLE;
```

Folgende Funktionen in *TDrawView* entsprechen den jeweiligen Funktionen in *TMyWindow* fast. Alle Modifikationen in den Funktionen sind in der rechten Tabellenspalte aufgeführt:

Funktion	TDrawView-Version
EvLButtonDown	Setzt das Flag *IsDirty* nicht. Darauf muß in EvLButtonUp geachtet werden.
EvRButtonDown	Keine Änderungen.
EvMouseMove	Keine Änderungen.
EvLButtonUp	Überprüft, ob die Maus verschoben wurde, nachdem die linke Maustaste gedrückt wurde. Wenn ja, dann wird die Dokumentfunktion *AddLine* aufgerufen, um den Punkt einzufügen.
CmPenSize	Keine Änderungen.
CmPenColor	Keine Änderungen.

Die Funktion *VnCommit* liefert immer den Wert TRUE zurück. In einer etwas komplexeren Anwendung würde diese Funktion alle aufgefangenen Daten in das Dokument einfügen. In unserer Anwendung werden die Daten jedoch in das Dokument eingefügt, wenn die jeweilige Zeile ausgegeben wird.

Die Funktion *VnRevert* erklärt den Ausgabebereich für ungültig, löscht ihn und gibt die Ausgabedaten im Fenster erneut aus. Der Rückgabewert dieser Funktion ist TRUE.

Definieren von Dokument-Templates

Haben Sie eine Dokumentklasse und eine zugehörige View-Klasse erzeugt, dann müssen Sie beide so miteinander verbinden, daß sie zusammenarbeiten können. Eine Verbindung zwischen einer Dokumentklasse und einer View-Klasse wird auch als Dokument-Templateklasse bezeichnet. Über eine Dokument-Templateklasse ermittelt der Dokumentmanager, welche View-Klasse für die Anzeige eines Dokuments geöffnet werden soll.

Eine Dokument-Templateklasse können Sie mit Hilfe des Makros DEFINE_DOC_TEMPLATE_CLASS erzeugen. An dieses Makro müssen drei Parameter übergeben werden. Der erste Parameter ist der Name der Dokument-Klasse, der zweite ist der Name der View-Klasse und der dritte ist der Name der Dokument-Templateklasse. Das Makro zur Erstellung einer Templateklasse könnte für die Klassen *TDrawDocument* und *TDrawView* zum Beispiel folgendermaßen aussehen:

```
DEFINE_DOC_TEMPLATE_CLASS(TDrawDokument, TDrawView, DrawTemplate);
```

Haben Sie eine Dokument-Templateklasse erstellt, dann müssen Sie eine Instanz dieser Klasse erzeugen. Als Name der Dokument-Templateklasse wird der Klassentyp verwendet. Der Instanz sollten Sie einen selbsterklärenden Namen geben.

Der Konstruktor für alle Dokument-Templateklassen besitzt folgende Syntax:

```
TplName name(const char far* desc,
             const char far* filt,
             const char far* dir,
             const char far* ext,
             long flags = 0);
```

Dabei ist:

- *TplName* der bei der Definition einer Templateklasse angegebene Klassenname.
- name der Name der Instanz.
- desc ist eine Textbeschreibung zu dem Template, die in den Dialogfenstern Datei öffnen und Speichern als Dateityp angezeigt wird.
- filt ist ein String, der als Filter für die Namen der Dateien aus dem aktuellen Verzeichnis verwendet wird; in diesem String kann ein beliebiger regulär gültiger DOS-Ausdruck enthalten sein.
- dir ist das Standardverzeichnis, in dem nach Dokumentdateien gesucht wird.
- ext ist die Standarderweiterung, die verwendet wird, wenn beim Speichern von Dateien keine spezielle Dateierweiterung angegeben wurde; wird hierbei 0 angegeben, dann bedeutet dies: keine Standarderweiterung.

- flags enthält den Modus, in dem das Dokument geöffnet oder erzeugt wurde; Dieser Wert kann einen oder mehrere der nachfolgend aufgeführten Flagwerte annehmen: *dtAutoDelete, dtNoAutoView, dtSingleView, dtAutoOpen, dtConfirm* oder *dtHidden*. Diese Flags sind im ObjektWindows-*Referenzhandbuch* sowie im Kapitel 11 des vorliegenden Handbuchs beschrieben.

Im folgenden Programmbeispiel wird gezeigt, wie eine Template-Instanz für die Klassen *TDrawDocument* und *TDrawView* aussehen könnte:

```
DrawTemplate drawTpl("Point Files (*.PTS)",
                     "*.pts", 0, "PTS",
                     dtAutoDelete|dtUpdateDir);
```

Unterstützung von Dok/View in einer Anwendung

In STEP11.CPP finden Sie den Quelltext zum Anwendungsobjekt sowie die Definition des Hauptfensters. Das Anwendungsobjekt enthält einen Framework für die in STEP11DV.CPP definierten Dok/View-Klassen. Im vorliegenden Abschnitt werden die Änderungen erläutert, die in der *TMyApp*-Klasse erforderlich sind, um die neuen Dok/View-Klassen unterstützen zu können. Die *OwlMain*-Funktion bleibt unverändert.

Die Funktion InitMainWindow

In der *InitMainWindow*-Funktion sind einige kleine Veränderungen erforderlich, damit das Dok/View-Modell unterstützt werden kann:

- Statt dem Konstruktor *TMyWindow* muß dem *TDecoratedFrame*-Konstruktor der Wert 0 für das Client-Rahmenfenster übergeben werden. Das Client-Fenster wird in der Funktion *EvNewView* gesetzt.

Der Aufruf von *AssignMenu* wird in einen *SetMenuDescr*-Aufruf umgeändert. Die von *TFrameWindow* abgeleitete Funktion *SetMenuDescr* bekommt nur einen Parameter, den Parameter *TMenuDescr,* übergeben. Das *TMenuDescr*-Objekt muß über die Menüressource COMMANDS aufgebaut werden. Dieser Aufruf besitzt folgendes Aussehen:

```
GetMainWindow()->SetMenuDescr(TMenuDescr("COMMANDS",1,0,0,0,0,1));
```

- Weiterhin wird ein Aufruf der Funktion *SetDocManager* hinzugefügt. Diese Funktion bestimmt das *DocManager*-Element der Klasse *TApplication*. Sie übernimmt *TDocManager* * als einzigen Parameter.
- Der *TDocManager*-Konstruktor erwartet einen Parameter mit einem oder mehreren Flags ORed. Das einzige unbedingt erforderliche Flag ist entweder *dmSDI* oder *dmMDI*. Über diese Flags wird der Dokumentmanager festgelegt, der eine dmSDI- (single-document interface) oder eine *dmMDI*-Anwendung (multiple-document interface) überwachen soll.

In unserem Beispiel wird eine SDI-Anwendung erzeugt. Sie müssen also das *dmSDI*-Flag angeben. Außerdem müssen Sie das *dmMenu*-Flag angeben, wodurch der Dokumentmanager sein Standardmenü bereitstellt.

Die Funktion *SetDocManager* muß folgendermaßen aufgerufen werden:

```
SetDocManager(new TDocManager(dmSDI | dmMenu));
```

Die Funktion InitInstance

Die *InitInstance*-Funktion wird modifiziert, weil viele verschiedene Funktionsaufrufe nach dem Erzeugen des Hauptfensters erfolgen müssen. *InitInstance* muß zuerst die *TApplication*-Version von *InitInstance* aufrufen. Diese Funktion ruft dann die Funktion *InitMainWindow* auf, die zuerst ein Hauptfensterobjekt erzeugt und danach das Hauptfenster selbst erstellt.

Nachdem die Basisklassenfunktion *InitInstance* aufgerufen wurde, müssen Sie noch die Hauptfensterfunktion *DragAcceptFiles* aufrufen und dabei den Parameter TRUE angeben. Dadurch kann das Hauptfenster Dateien entgegennehmen, die im Fenster abgelegt werden. Die Funktionalität Ziehen und Ablegen wird über die Antworttabelle der Anwendung erreicht (siehe nächster Abschnitt).

Um dem Benutzer die Ausgabe in das Fenster gleich zu Beginn (also bei Start der Anwendung) zu ermöglichen, müssen Sie die Funktion *CmFileNew* des Dokumentmanagers aufrufen. Diese Funktion erzeugt im Hauptfenster ein neues unbenanntes Dokument samt View.

Die *InitInstance*-Funktion sollte in etwa wie folgt aussehen:

```
void
TMyApp::InitInstance()
{
  TApplication::InitInstance();
  GetMainWindow()->DragAcceptFiles(TRUE);
  GetDocManager()->CmFileNew();
}
```

Funktionen zu TMyApp hinzufügen

Die *TMyApp*-Klasse fügt mehrere neue Funktionen ein. Sie überschreibt die *TApplication*-Version von *InitInstance*. Weiterhin fügt sie eine Antworttabelle ein und übernimmt die Funktion *CmAbout* von der *TMyWindow*-Klasse. Sie ermöglicht das Ziehen und Ablegen durch Hinzufügen des Makros EV_WM_DROPFILES in die Antworttabelle und fügt die Funktion *EvDropFiles* ein, um das Ereignis zu verarbeiten. Weiterhin wird das neue Ereignis WM_OWLVIEW verarbeitet. Dieses Ereignis deutet auf eine View-Anfragebotschaft hin. Zwei Funktionen bearbeiten diese Botschaft. *EvNewView ver*arbeitet eine solche WM_OWLVIEW-Botschaft über den Parameter *dnCreate*. *EvCloseView* verarbeitet eine WM_OWLVIEW-Botschaft über den Parameter *dnClose*.

Im folgenden Programmbeispiel finden Sie die neue Deklaration der *TMyApp*-Klasse zusammen mit der Definition der Antworttabelle:

```
class TMyApp : public TApplication {
  public:
    TMyApp() : TApplication() {}

  protected:
    // Methoden von TApplication modifizieren
    void InitInstance();
    void InitMainWindow();

    // Ereignisbehandlung
    void EvNewView  (TView& View);
    void EvCloseView(TView& View);
    void EvDropFiles(TDropInfo dropInfo);
    void CmAbout();
  DECLARE_RESPONSE_TABLE(TMyApp);
};

DEFINE_RESPONSE_TABLE1(TMyApp, TApplication)
  EV_OWLVIEW(dnCreate, EvNewView),
  EV_OWLVIEW(dnClose,  EvCloseView),
  EV_WM_DROPFILES,
  EV_COMMAND(CM_ABOUT, CmAbout),
END_RESPONSE_TABLE;
```

Die Funtkion CmAbout

Die *CmAbout*-Funktion entspricht der gleichnamigen Funktion aus *TMyWindow* beinahe. Der einzige Unterschied besteht darin, daß die *CmAbout*-Funktion nicht mehr länger in ihrer übergeordneten Fensterklasse enthalten ist. Der *this*-Zeiger wird durch einen Aufruf der Funktion *GetMainWindow* ersetzt. Die Funktion müßte nun folgendermaßen aussehen:

```
void TMyApp::CmAbout()
{
  TDialog(GetMainWindow(), IDD_ABOUT).Execute();
}
```

Die Funktion EvDropFiles

Die Funktion *EvDropFiles* verarbeitet das Ereignis WM_DROPFILES. Diese Funktion erwartet ein *TDropInfo*-Objekt als Parameter. Dieses Objekt enthält Funktionen zur Ermittlung der Anzahl der abgelegten Dateien, der Namen dieser Dateien, der Positionen, an denen die jeweiligen Dateien abgelegt wurden u.s.w.

Weil es sich in unserem Beispiel um eine SDI-Anwendung handelt, müssen Sie dem Benutzer mitteilen, daß nur nur eine Anwendung gleichzeitig abgelegt werden darf, wenn die Anzahl der Dateien größer als Eins ist. Zur Ermittlung der Anzahl der abgelegten Dateien können Sie die TDropInfo-Funktion *DragQueryFileCount* aufrufen. Dieser Funktion müssen keine Parameter übergeben werden. Sie gibt die Anzahl der abgelegten Dateien zurück. Wenn der Dateizähler größer als Eins ist, dann sollten Sie ein Meldungsfenster mit einer Warnung öffnen.

Nun benötigen Sie den Namen der abgelegten Datei. Sie können die Länge des Dateipfad-Strings mit Hilfe der *TDropInfo*-Funktion *DragQueryFileNameLen* herausfinden. Diese Funktion erwartet einen Parameter mit der Angabe des Datei-Index, über den die Untersuchung erfolgt. Da Sie ja wissen, das nur eine Datei abgelegt worden sein kann, muß dieser Parameter 0 sein. Die Funktion liefert die Länge des Dateipfades zurück.

Belegen Sie Speicherplatz für einen String der erforderlichen Länge und rufen Sie danach die *TDropInfo*-Funktion *DragQueryFile* auf. Dieser Funktion müssen drei Parameter übergeben werden. Der erste ist der Index der Datei. Dieser Parameter sollte wieder 0 sein. Der zweite Parameter ist ein **char *** für den Dateipfad. Der dritte enthält die Länge des Dateipfades. Diese Funktion fügt den Dateipfad in das **char** Array (zweiter Parameter).

Haben Sie den Dateinamen ermittelt, müssen Sie noch das richtige Template für den Dateityp herausfinden. Dazu sollten Sie die Dokumentmanager-Funktion *MatchTemplate* aufrufen. Diese Funktion durchsucht die im Dokumentmanager enthaltene Liste der Dokument-Templates und liefert einen Zeiger auf das erste Template zurück, dessen Muster zur abgelegten Datei paßt. Dieser Zeiger ist ein *TDocTemplate **. Wenn der Dokumentmanager kein passendes Template finden konnte, dann ist der Rückgabewert gleich 0.

Haben Sie ein Template gefunden, dann rufen Sie die Template-Funktion *CreateDoc* auf und übergeben den Dateipfad als Parameter an die Funktion. Dadurch wird ein neues Dokument und sein zugehöriges View erzeugt und die angegebene Datei in dem Dokument geöffnet.

Ist die Datei geöffnet, dann müssen Sie unbedingt die Funktion *DragFinish* aufrufen. Diese Funktion gibt den Speicher wieder frei, den Windows während einer Zieh-und-Ablege-Operation belegt hat.

Im folgenden Programmbeispiel können Sie sehen, wie die Funktion *EvDropFiles* aussehen sollte:

```
void
TMyApp::EvDropFiles(TDropInfo dropInfo)
{
  if (dropInfo.DragQueryFileCount() != 1)
    ::MessageBox(0,"Can only drop 1 file in SDI mode","Drag/Drop
Error",MB_OK);
  else {
    int fileLength = dropInfo.DragQueryFileNameLen(0)+1;
    char* filePath = new char [fileLength];
    dropInfo.DragQueryFile(0, filePath, fileLength);
    TDocTemplate* tpl = GetDocManager()->MatchTemplate(filePath);
    if (tpl)
      tpl->CreateDoc(filePath);
    delete filePath;
  }
```

```
  dropInfo.DragFinish();
}
```

Die Funktion EvNewView

Die Botschaft WM_OWLVIEW informiert die Anwendung, wenn in einem View ein Ereignis auftritt. Alle Funktionen, die die Botschaft WM_OWLVIEW verarbeiten, geben **void** zurück. Sie erwarten einen *TView* **&**-Parameter. Ist *dnCreate* der Parameter dieses Ereignisses, dann bedeutet dies, daß ein neues View-Objekt erzeugt wurde und die Anwendung auffordert, das View-Fenster einzurichten.

In unserem Beispiel müssen Sie das View-Fenster als Client des Hauptfensters festlegen. Dafür sind zwei Funktionen vorhanden: *GetWindow* und *SetClientWindow*.

Die Funktion *GetWindow* ist ein Element der View-Klasse. Sie nimmt keine Parameter entgegen. Der Rückgabewert ist ein *TWindow* *****. Dieser zeigt auf das View-Fenster.

Haben Sie sich den Zeiger auf das View-Fenster ausgeben lassen, dann können Sie dieses Fenster mit Hilfe der Hauptfenster-Funktion *SetClientWindow* als Client-Fenster definieren. Dieser Funktion muß ein *TWindow* ***** als Parameter übergeben werden. Sie deklariert das Fensterobjekt als Client-Fenster. Rückgabewert ist ein *TWindow* *****. Dieser Zeiger zeigt auf das alte Client-Fenster, wenn vorhanden.

Bevor Sie weiterarbeiten, sollten Sie überprüfen, ob das neue Client-Fenster erfolgreich erzeugt wurde. In der Klasse *TView* ist die *IsOK*-Funktion enthalten. Diese gibt FALSE zurück, wenn ein Fenster nicht erfolgreich erzeugt wurde. Wenn also *IsOK* FALSE zurückgibt, dann sollten Sie den Aufruf von *SetClientWindow* wiederholen, dabei aber 0 als Fenster-Zeiger angeben.

Wenn das Fenster erfolgreich erzeugt wurde, dann müssen Sie über die Funktion *GetViewMenu* das View-Menü suchen. Wenn das View ein Menü besitzt, dann können Sie mit Hilfe der Funktion *MergeMenu* das View-Menü mit dem Menü des Fensters verschmelzen.

Es folgt ein Codefragment als Vorschlag zur Funktion *EvNewView*:

```
void
TMyApp::EvNewView(TView& View)
{
  GetMainWindow()->SetClientWindow(View.GetWindow());
  if (!View.IsOK())
    GetMainWindow()->SetClientWindow(0);
  else if (View.GetViewMenu())
    GetMainWindow()->MergeMenu(*View.GetViewMenu());
}
```

Die Funktion EvCloseView

Wurde *dnClose* als Parameter für das Ereignis WM_OWLVIEW angegeben, dann bedeutet dies, daß ein View geschlossen wurde. Auf diesen Fall wird mit dem Parameter *EvCloseView* reagiert. Wie die Funktion *EvNewView* gibt auch die Funktion *EvCloseView* **void** zurück und benötigt einen *TView* **&**-Parameter.

Soll ein View geschlossen werden, dann müssen Sie das View-Fenster als Client des Hauptfensters deaktievieren. Dazu rufen Sie die Hauptfenster-Funktion *SetClientWindow* auf und übergeben diesem als Fensterzeiger eine 0. Danach stellen Sie mit Hilfe der Hauptfensterfunktion *RestoreMenu* das Menü des Rahmenfensters in seinem früheren Status wieder her.

Wenn über *EvNewView* ein neues View erzeugt wird, dann wird die Überschrift des Rahmenfensters auf den Dateipfad des Dokuments gesetzt. Sie müssen also die Überschrift des Hauptfensters über die Funktion *SetCaption* zurücksetzen.

Es folgt ein Programmbeispiel zu der Funktion *EvCloseView*:

```
void
TMyApp::EvCloseView(TView& /*View*/)
{
  GetMainWindow()->SetClientWindow(0);
  GetMainWindow()->RestoreMenu();
  GetMainWindow()->SetCaption("Drawing Pad");
}
```

Weiterführende Informationen

Hier ist nun eine Aufstellung darüber, wo Sie weitere Informationen zu den in diesem Schritt eingeführten Themen finden können:

- Die Dok/View-Klassen sind in Kapitel 11 beschrieben.
- Eine Beschreibung zu den Menü-Objekten und Menü-Deskriptoren finden Sie in Kapitel 9 sowie im *ObjektWindows-Referenzhandbuch*.
- Die Funktionen *InitMainWindow* und *InitInstance* werden in Kapitel 3 ausführlich behandelt.
- Eine Erläuterung zu den Funktionen Ziehen und Ablegen finden Sie im *ObjektWindows-Referenzhandbuch*.

Schritt 13: Erstellen eines OLE-Containers

Der nächste Schritt zeigt, wie aus der Zeichenbrett-Anwendung ein OLE 2-Container zu machen ist. Objektverknüpfung und einbettung (Object Linking and Embedding, OLE) ist eine Erweiterung von Windows, durch die der Benutzer diverse Anwendungen in einem Arbeitsbereich nahtlos kombinieren kann. Eine OLE-Containeranwendung kann mehrere Dienstobjekte versorgen, indem sie zusätzlichen Arbeitsbereich zur

Verfügung stellt, in dem der Benutzer Ihre Anwendung um die Fähigkeiten der OLE-fähigen Anwendung erweitern kann.

Der Code des in diesem Schritt verwendeten Beispiels befindet sich in den Dateien step14.cpp, step14dv.cpp, step14.rc und step14dv.rc im EXAMPLES\OWL\TUTORIAL-Verzeichnis, wo Ihr Compiler installiert ist.

Wie OLE funktioniert

Für die grundlegenden OLE-Operationen sind zwei verschiedene Typen von Anwendungen nötig:

- Ein *Container* kann in sich andere Anwendungen oder Objekte eingebettet haben, wobei die Daten der eingebetteten Objekte als Teil der dem Container eigenen Datenmenge präsentiert werden.
- Ein *Server* kann in einer Container-Anwendung eingebettet sein und dazu benutzt werden, die im Arbeitsbereich des Containers angezeigten Daten zu manipulieren.

Was ist ein Container?

In diesem Schritt verwandeln Sie ihre Dok/View-Zeichenbrett-Anwendung in einen OLE-Container. Das Zeichenbrett zu einem OLE-Container zu machen, bringt einige beachtliche Erweiterungen: Die Anwendung ist nicht mehr auf die Darstellung einer Menge von Linien beschränkt, sondern kann mittels eines von einem Benutzer in dessen Abbildungen eingebetteten Servers jede Art von Daten anzeigen, die dieser Server präsentieren kann. Ungeachtet dessen, daß die Fähigkeit für Linienzeichnungen bestehen bleibt und dies auch weiterhin die Hauptaufgabe der Anwendung ist, können Benutzer nun ihre Abbildungen mit Bitmaps, Arbeitsblatt-Diagrammen und sogar Tondateien würzen.

Obwohl Sie viele derartige Aufgaben mit Hilfe der Zwischenablage zum Übertragen von Daten erledigen können, ist es doch einfacher, OLE zu verwenden. Um die Zwischenablage zu verwenden, muß Ihre Anwendung in der Lage sein, die dort gespeicherten Daten zu bearbeiten. Mit anderen Worten, falls Sie in Ihrer Anwendung Bitmaps tolerieren wollen, sind Funktionen zum Entgegennehmen und Anzeigen von Bitmaps vonnöten. Damit kann die Anwendung aber noch lange nichts mit Arbeitsblatt-Diagrammen, Datenbank-Tabellen oder Daten anderer graphischer Formate anfangen. Um weitere Datentypen hinzuzunehmen, müssen weitere Funktionen zum Interpretieren und Anzeigen dieser Daten implementiert werden.

Bei Verwendung von OLE kann Ihre Anwendung jede Art von Daten anzeigen, die von irgendeinem verfügbaren OLE-Server unterstützt wird. Für Ihre Anwendung sieht ein Bitmap dann genauso aus wie ein Arbeitsblatt-Diagramm, eine Datenbank-Tabelle oder jede andere Art von Objekten, d.h. sie sehen alle wie ein OLE-Server-Objekt aus.

Sie können natürlich ihrer Anwendung beibrigen, unter Verwendung der Zwischenablage ein Bitmap anzuzeigen. Aber nach dem Übernehmen des Bitmaps müssen Sie dann gegebenenfalls zusätzliche Funktionen zur Modifizierung desselben einbauen.

Bei Verwendung von OLE handhabt der eingebettete Server die Daten, sobald der Benutzer sie irgendwie ändern will. Die im Server bearbeiteten Datentypen haben keinerlei Auswirkung auf den Container.

Implementierung von OLE in ObjectWindows: ObjectComponents

Die Vorteile, die OLE ihrer Anwendung bringen, kosten etwas: Historisch gesehen war die Programmierung von OLE-Implementationen sehr mühsam und zeitaufwendig. Der Code mußte gemäß den OLE-Spezifikationen modifiziert werden, und darüber hinaus folgt OLE nichtmal den ereignisbasierten Paradigmen, auf denen Windows-Anwendungen ehemals beruhten. Stattdessen implementiert OLE ein neues schnittstellenbasiertes Paradigma, wozu ein Verständnis der Standard-OLE-Schnittstellen, des Referenzzählers und anderer OLE-Spezifikationen nötig ist.

ObjectWindows implementiert OLE mit Hilfe des ObjectComponents-Gerüstes. Unter Verwendung von ObjectComponents können Sie Ihre Anwendung mit nur geringen Code-Änderungen zu einem OLE-Container oder Server machen. ObjectComponents kann in Verbindung mit folgenden Arten von Anwendungen benutzt werden:

- Dok/View-ObjectWindows
- nicht-Dok/View-ObjectWindows
- nicht-ObjectWindows-C++

Wie wir in diesem Schritt noch sehen werden, sind die wenigsten Änderungen bei Dok/View-ObjectWindows-Anwendungen nötig. Wie OLE bei Anwendungen der Sorte nicht-Dok/View-ObjectWindows bzw. nicht-ObjectWindows-C++ zu implementieren ist, ist in den Kapiteln 18 bis 23 des *ObjectWindows Programmierhandbuchs* beschrieben.

Die Umwandlung Ihrer Dok/View-ObjectWindows-Anwendung in eine OLE-Containeranwendung verlangt folgende Schritte:

- Einfügen der passenden Header-Dateien
- Registrierung Ihrer Anwendung und der Dok/View-Objekte in der Registrierungsdatenbank des Systems
- Erzeugen eines *TOcApp*-Objektes und dessen Assoziierung mit Ihrem Anwendungsobjekt
- Ändern der Rahmenfenster-Klasse in eine OLE-bewußte Rahmenfenster-Klasse
- Ändern der Basisklassen der Dokument und Viewklasse in OLE-fähige Klassen

Die in diesem Schritt verwendeten ObjectComponents-Objekte werden jeweils beim Hinzufügen zur Zeichenbrett-Anwendung erklärt. Das ObjectComponents-Gerüst ist detailliert in den Kapiteln 18 bis 23 des *ObjectWindows Programmierhandbuchs* und im *ObjectWindows Referenzhandbuch* beschrieben.

Hinzufügen von Header-Dateien für OLE-Klassen

Um ObjectComponents-Klassen verwenden zu können, müssen Sie ihren Dateien neue Header hinzufügen. ObjectComponents fügt die OLE-Fähigkeiten durch Ableiten neuer OLE-fähiger Klassen aus existierenden Klassen hinzu.

Für das Hinzufügen neuer Header zu Ihren Dateien können Sie ObjectComponents-Klassen verwenden:

In step14.cpp verwenden Sie *TOleMDIFrame* – einen OLE-fähigen dekorierten MDI-Rahmen – statt des *TDecoratedMDIFrame*. Alle OLE-Rahmenfenster, egal ob MDI oder SDI, müssen Dekorationen handhaben können, da viele eingebettete OLE-Server ihre eigene Mauspalette mitbringen.Die Klasse *TOleMDIFrame* ist in der Header-Datei owl\olemdifr.h deklariert.

Ihre Liste von Include-Anweisungen sollte nun etwa wie folgt aussehen:

```
#include <owl/applicat.h>
#include <owl\dialog.h>
#include <owl\controlb.h>
#include <owl\buttonga.h>
#include <owl\statusba.h>
#include <owl\docmanag.h>
#include <owl\olemdifr.h>
#include <stdlib.h>
#include <string.h>
#include "step14.rc"
```

In step14dv.cpp müssen Sie die OLE-fähigen Dokument und View-Klassen einfügen. Diese Klassen – *TOleDocument* und *TOleView* – bieten die standardmäßige Dok/View-Funktionalität sowie die Unterstützung von OLE. Sie sind in den Header-Dateien owl\oledoc.h bzw owl\oleview.h deklariert. Ihre Liste von Include-Anweisungen in step14dv.cpp müßte nun folgendermaßen aussehen:

```
#include <owl\chooseco.h>
#include <owl\dc.h>
#include <owl\docmanag.h>
#include <owl\gdiobjec.h>
#include <owl\inputdia.h>
#include <owl\listbox.h>
#include <owl\oledoc.h>
#include <owl\oleview.h>
```

```
#include <classlib\arrays.h>
#include "step14dv.rc"
```

Registrieren der Anwendung für OLE

Da OLE über alle in einem bestimmten System laufenden Anwendungen buchführt, muß jede Anwendung, die OLE verwenden möchte, sich in einer systemweiten OLE-Registrierungsdatenbank anmelden. Sie müssen eine eindeutige Identifizierungsnummer und eine Beschreibung der Anwendung liefern. Außerdem müssen Sie Objekte erzeugen, die die Anwendung zur Kommunikation mit OLE befähigen.

ObjectComponents vereinfacht die Registrierung Ihrer Anwendung durch eine Menge von Registriermakros. Diese Makros erzeugen ein Objekt vom Typ *TRegList* – eine sogenannte Registriertabelle –, welches die von der OLE-Registrierungsdatenbank verlangten Informationen beinhaltet. Die Makros sind dieselben wie die, die Sie beim Erzeugen eines Dok/View-Templates benutzen; Sie verwenden hier aber umfangreichere Fähigkeiten der *TRegList*-Klasse.

Sobald Sie eine Registriertabelle erzeugt haben, müssen Sie diese einem Verbinderobjekt übergeben. Ein Verbinderobjekt stellt den Kanal zur Verfügung, durch den eine Dok/View-Anwendung mit ObjectComponents und (über Erweiterungen) mit OLE kommuniziert. Die Registriertabelle wird einem Objekt vom Typ *TOcApp* übergeben. (ObjectComponents-Verbinderobjekte beginnen alls mit *TOc*.)

Später werden Sie die Deklaration Ihrer *TDrawApp*-Klasse modifizieren, so daß diese sowohl von *TApplication* als auch von *TOcModule* abgeleitet ist. Ihr Anwendungsobjekt initialisiert das *TOcApp*-Verbinderobjekt während des Aufbaus der Objekte der Anwendung. Auf das Verbinderobjekt wird dann mittels eines Zeigers aus der *TOcModule*-Klasse zugegriffen.

Erzeugen einer Registriertabelle

Sie erzeugen mit Hilfe des Makros REGDATA eine Registriertabelle für Ihre Container-Anwendung. Dies ist dasselbe Makro, das sie schon zuvor zur Registrierung der Standarddateinamenserweiterung Ihrer Dokumente und Ihres Dateinamen-Filters verwendet haben. Für die jetzigen Zwecke benötigen Sie folgende Schlüsselwerte:

Tabelle 14.1: Schlüsselwerte und ihre Bedeutung

Schlüsselwert	Bedeutung
clsid	String-Darstellung einer 16-Byte-Zahl, die als global eindeutiger ID oder GUID bezeichnet wird. Diese Zahl muß eindeutig für Ihre Anwendung sein; sie dient der Unterscheidung Ihrer Anwendung von allen anderen Anwendungen im System. Der Wert ist nur für internen Gebrauch im System bestimmt.
description	Eine für den Systembenutzer bestimmte Beschreibung der Anwendung. Der String erscheint in der OLE-Registrierungsliste.

Ihre Registriertabelle sollte etwa von folgender Form sein:

```
REGISTRATION_FORMAT_BUFFER(100)

BEGIN_REGISTRATION(AppReg)
    REGDATA(clsid, "{383882A1-8ABC-101B-A23B-CE4E85D07ED2}")
    REGDATA(description,"OWL Drawing Pad 2.5")
END_REGISTRATION
```

Sie müssen einen für Ihre Anwendung eindeutigen GUID wählen. Es gibt verschiedene Möglichkeiten, einen solchen zu finden. Das Erzeugen eines GUID und der Anwendungsbeschreibung ist detailliert in Kapitel 20 des *ObjectWindows Programmer's Guide* dargestellt. Für diesen Schritt können Sie die in den Schritt-Beispielen gegebenen GUIDs benutzen. Verwenden Sie aber nicht dieselben Nummern, wenn Sie andere Anwendungen erzeugen.

Auch andere Makros können in Ihre Registriertabelle eingehen. Die zum Erzeugen von *AppReg* bilden das absolute Minimum für ein Container-Anwendungsobjekt. Sie werden noch eine kompliziertere Tabelle bei der Erzeugung der Registriertabelle für Ihre Dokumentklasse zu sehen kriegen.

Darüber hinaus ist es sicherer – da *AppReg* im globalen Namensbereich Ihrer Anwendung erzeugt wird – und aufschlußreicher, von innerhalb der Klassen und Funktionen unter Verwendung der globalen Zugriffsoperators darauf zuzugreifen. Sie sollten also an Stelle von

```
void
MyClass::MyFunc()
{
    OtherFunc(AppReg);
}
```

folgendes schreiben:

```
void
MyClass::MyFunc()
{
    OtherFunc(::AppReg);
}
```

Erzeugen einer Klassenfabrik

Eine Klassenfabrik ist so ungefähr das, wonach es klingt – ein Objekt, das weitere Objekte machen kann. Es wird in OLE dazu verwendet, Objekte für Verknüpfung und Einbettung bereitzustellen. Wünscht eine Anwendung Ihr Anwendungsobjekt in sich einzubetten, so ist es die Klassenfabrik die das eingebettete Objekt erzeugt.

Unter ObjectWindows läßt sich eine Klassenfabrik leicht mit Hilfe des *TOleDocViewFactory*-Templates erzeugen. Sie brauchen lediglich eine Instanz des Templates mit Ihrer Anwendungsklasse als Template-Typ zu erzeugen. Im hiesigen Fall wollen Sie eine Instanz von *TDrawApp* mit Ihrer Fabrik erzeugen. Das Erstellen des Templates sieht wie folgt aus:

```
TOleDocViewFactory<IDrawApp>();
```

Eine Instanz dieses Templates ist als zweiter Parameter dem Konstruktor *TOcRegistrar* zu übergeben. Wie dies aussieht, können Sie dem noch folgenden Beispiel *OwlMain* entnehmen. Die eigentlichen Objekte werden in der Fabrik unter Verwendung der gleichen Dok/View-Templates erzeugt, die auch von Ihrer eigenständig laufenden Anwendung benutzt wurden.

TOleViewFactory ist das Klassenfabrik-Template für Dok/View-ObjectWindows-Anwendungen. Für andere Anwendungstypen gibt es andere Klassenfabrik-Templates, welche in Kapitel 19 des *ObjectWindows Programmierhandbuchs* und im *ObjectWindows Referenzhandbuch* diskutiert werden.

Erzeugen eines Registrarobjektes

Die Registriertabelle enthält für das System gesachte Informationen über Ihre Anwendung. Das Registrarobjektes (vom Typ *TOcRegistrar*) nimmt die Registriertabelle und registriert die Anwendung in der OLE-Registrierungsdatenbank. Außerdem zerlegt es die Kommandozeile der Anwendung auf der Suche nach mit OLE in Zusammenhang stehenden Optionen.

Ein Registrarobjekt wird wie folgt erzeugt:

Zunächst ist unter Verwendung des *TPointer*-Templates, welches in der Header-Datei osl\geometry.h definiert ist, ein globaler statischer Zeiger auf ein *TOcRegistrar*-Objekt zu erzeugen. (Die Datei owl\geometry.h ist bereits von verschiedenen ObjectWindows-Header-Dateien eingebunden, so daß Sie sie folglich nicht erneut einbinden müssen.) Dies sieht dann wie folgt aus:

```
static TPointer<TOcRegistrar> Registrar;
```

Die Verwendung von *TPointer* an Stelle eines einfachen Zeigers, wie etwa *TOcRegistrar* Registrar*, gewährleistet automatisches Löschen, wenn das bezeichnete Objekt entfernt wird oder außer Reichweite gerät. In *Tpointer* ist die gesamte Bandbreite von mit ordentlichen Zeigern möglichen Operationen verfügbar; darüber hinaus sind aber einige der traditionellen Gefahren bei der Benutzung von Zeigern eliminiert.

Erzeugen Sie das eigentliche Registrarobjekt. Der Konstruktor *TOcRegistrar* nimmt vier Parameter entgegen:

- einen Verweis auf ein Registriertabellenobjekt
- einen Zeiger auf eine Callback-Funktion vom Typ *TComponentCreate*
- ein die Kommandozeile der Anwendung enthaltender String
- ein Instanz-Handle, das die Anwendungsinstanz bezeichnet, für die der Registrar gedacht ist (standardmäßig *_hInstance*, also die aktuelle Anwendungsinstanz)

Bei der Erstellung des Registrarobjektes können Sie für diese Parameter nachfolgende Argumente übergeben:

- Übergeben Sie für den ersten Parameter Ihr Registriertabellenobjekt.
- Für den zweiten Parameter übergeben Sie Ihre Klassenfabrik.
- Für den dritten Parameter ist die Kommandozeile der Anwendung zu übergeben. Sie erhalten diese durch Aufruf der Funktion *GetCmdLine* von *TApplication*.
- Den vierten Parameter, ein Instanz-Handle, brauchen Sie nicht zu spezifizieren; der ist mit seinem Vorgabewert gut bedient.

Ein Beispiel dafür:

```
::Registrar = new TOcRegistrar(AppReg, TOleDocViewFactory<TDrawApp>(),
                                                    TApplication::GetCmdLine());
```

Rufen Sie die Funktion *Run* auf, aber nicht die des Anwendungsobjektes (was auch gar nicht möglich wäre, da Sie noch kein Anwendungsobjekt erzeugt haben), sondern die des Registrarobjektes. *TOcRegistrar* bietet eine *Run*-Funktion, die ganz wie die von *TApplication* aufgerufen wird. Nichtsdestotrotz sollte jede ObjectWindows-OLE-Anwendung die *Run*-Funktion des Registrarobjektes aufrufen, welche einige Überprüfungen und für die OLE-Anwendung nötige Arbeiten durchführt.

Ihre Funktion *OwlMain* müßte nun folgendermaßen aussehen:

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
    ::Registrar = new TOcRegistrar(AppReg, TOleDocViewFactory<TDrawApp>(),

TApplication::GetCmdLine());
    return ::Registrar->Run();
}
```

Erzeugen eines Anwendungs-Dictionarys

Das Anwendungs-Dictionary ist ein Objekt, das bei der Koordinierung der Verbindungen zwischen Prozessen oder Tasks und den *TApplication*-Zeigern hilft. Bevor es OLE gab, war dies relativ einfach: Ein *TApplication*-Objekt war so ziemlich synonym zu einem Prozeß. Mit OLE gerät die Umgebung durcheinander: Dort können mannigfaltige Tasks und Prozesse in einer einzigen Anwendung sein, mit einer Container-Anwendung auch diverse eingebettete Server, eventuell in diesen noch mehr eingebettete Server – die Nachbarschaft ist eben etwas stärker bevölkert.

Um damit klarzukommen, liefert ObjectWindows mit der Klasse *TAppDictionary* Anwendungs-Dictionarys. Das Beste an *TAppDictionary* ist, daß Sie nicht viel darüber zu wissen brauchen, um es hier für unsere Zwecke verwenden zu können. ObjectWindows stellt auch ein Makro namens DE-

FINE_APP_DICTIONARY zur Verfügung, das für Sie ein Anwendungs-Dictionary erzeugt und initialisiert.

DEFINE_APP_DICTIONARY braucht einen Parameter, nämlich den Namen des Objektes, das Sie erzeugen wollen. Sie sollten dieses ziemlich an den Anfang Ihrer Quelltextdatei in den globalen Namensbereich plazieren. Zumindest muß es vor dem *TDrawApp*-Konstruktor stehen, weil es dort verwendet wird.

Die Definition Ihres Anwendungs-Dictionarys sollte etwa von folgender Form sein:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

Ändern von *TDrawApp*

Sie müssen die Klasse *TDrawApp* für die Unterstützung von ObjectComponents abändern. Diese Änderungen sind eine Art Standard, wenn Sie in einem OLE-Container eine Dok/View-Anwendung erzeugen.

- Ändern der Klassendeklaration
- Ändern der Klassenfunktionalität einschließlich
  - Erzeugung eines OLE-MDI-Rahmens
  - Setzen des mit dem OLE-MDI-Rahmen assoziierten *TOcApp*
  - Hinzufügen eines Bezeichners für eine Mauspalette

Ändern der Klassendeklaration

An der Deklaration der Klasse *TDrawApp* sind folgende Änderungen vorzunehmen:

TDrawApp muß sowohl von der Klasse *TApplication* als auch von *TOcModule* abgeleitet sein. *TOcModule* liefert mittels des ObjectComponents-Gerüstes die von Ihrem Anwendungsobjekt zur Kommunikation mit OLE verwendete Schnittstelle. Sowohl *TApplication* als auch *TOcModule* sollte eine öffentliche Basisklasse sein.

Ändern Sie den Konstruktor so, daß Sie dem *TApplication*-Konstruktor einen einzigen Parameter übergeben. Sie sollten den Namen des Anwendungsobjektes mit dem Wert aus *description* von *AppReg* initialisieren. Um dies zu vereinfachen, überladet die Klasse *TRegList* den []-Operator so, daß der mit dem Schlüsselwert, welcher zwischen den eckigen Klammern übergeben wurde, assoziierte String zurückgegeben wird. Also brauchen Sie nur `AppReg["description"]` aufzurufen, um den mit dem Schlüssel *description* assoziierten String zu erhalten.

Ihre *TDrawApp*-Deklaration sollte nun folgendem Code ähneln:

```
class TDrawApp : public TApplication, public TOcModule {
    public:
         TDrawApp() : TApplication(::AppReg["description"]) {}
```

```
    protected:
        TMDIClient* Client;
        // Überschreiben von Methoden von TApplication
        void InitInstance();
        void InitMainWindow();

        // Behandlung von Ereignisse
        void EvNewView(TView& view);
        void EvCloseView(TView& view);
        void EvDropFiles(TDropInfo dropInfo);
        void CmAbout();

    DECLARE_RESPONSE_TABLE(TDrawApp);
};
```

Ändern der Klassenfunktionalität

Sie müssen aus dem Hauptfenster ein *TOleMDIFrame*-Objekt machen und es wie folgt passend initialisieren:

- Erzeugen eines OLE-MDI-Rahmens
- Setzen des mit dem OLE-MDI-Rahmen assoziierten *TOcApp*
- Hinzufügen eines Bezeichners für eine Mauspalette

Erzeugen eines OLE-MDI-Rahmens

Als nächstes müssen Sie Ihre Funktion *InitMainWindow* ändern, indem Sie aus dem dekorierten MDI-Rahmen des Rahmenfenster-Objekts einen OLE-bewußten dekorierten MDI-Rahmen machen. (Beachten Sie, daß alle OLE-bewußten ObjectWindows-Rahmenfenster-Klassen dekoriert sind.) Die dafür zu verwendende Fensterklasse ist *TOleMDIFrame*. *TOleMDIFrame* basiert auf den Klassen *TMDIFrame*, welche MDI-Unterstützung bietet, sowie *TOleFrame*, welche die Fähigkeit zur Arbeit mit Object-Components liefert. Nachfolgend sehen Sie den Konstruktor für *TOleMDIFrame*:

```
TOleMDIFrame(const char far* title,
                  TResId menuResId,
                  TMDIClient& clientWnd = *new TMDIClient,
                  bool trackMenuSelection = false,
                  TModule* module = 0);
```

Die Parameter für den *TOleMDIFrame*-Konstruktor sind die gleichen wie für *TDecoratedMDIFrame*. Daher ist die Konvertierung einfach: Sie brauchen beim Erzeugen des Rahmenfenster-Objektes lediglich den Namen der Klasse zu ändern.

Setzen des mit dem OLE-MDI-Rahmen assoziierten *TOcApp*

Damit der OLE-MDI-Rahmen eingebettete OLE-Objekte handhaben kann, muß er wissen, wie die Kommunikation mit dem ObjectComponents-Mechanismus verläuft. Auf letzteren wird über das mit dem Anwendungsobjekt assoziierte *TOcApp*-Objekt zugegriffen, womit explizit ein Rahmenfenster zu assoziieren ist.

Dafür bietet *TOleMDIFrame* eine von *TOleFrame* geerbte Funktion namens *SetOcApp*, welche einen Zeiger auf ein *TOcApp*-Objekt entgegennimmt und **void** zurückgibt. Als Parameter können Sie *OcApp* an *SetOcApp* übergeben.

Hinzufügen eines Bezeichners für eine Mauspalette

OLE-Server ersetzen während ihrer Arbeit häufig Ihre Mauspalette durch eine selbst mitgebrachte. Die dazu nötigen Mechanismen werden von ObjektComponents bereitgestellt. Allerdings muß ObjektWindows erstmal Ihre Mauspalette finden können, um selbige durch eine Mauspalette des Servers ersetzten zu können.

ObjektWindows sucht nach Ihrer Mauspalette, indem es die Liste der zum OLE-MDI-Rahmenfenster gehörigen untergeordneten Fenster durchsucht und jeden einzelnen Fensterbezeichner überprüft. Bis jetzt hatte Ihre Mauspalette überhaupt keinen Bezeichner, und folglich würde ObjektWindows Ihre Mauspalette nicht finden. Damit ObjektWindows die Mauspalette des Containers identifizieren kann, muß der Container IDW_TOOLBAR als seinen Fensterbezeichner, also als *Id*-Element des *Attr*-Elementobjektes der Mauspalette, benutzen.

Nun sollte Ihre *InitMainWindow*-Funktion etwa folgendermaßen aussehen:

```
void
TDrawApp::InitMainWindow()
{
    // Erstellen eines OLE-fähigen MDI-Rahmens
    TOleMDIFrame* frame;
    frame = new TOleMDIFrame(GetName(), 0, *(Client = new TMDIClient),
true);

    // OcApp des Rahmens auf OcApp setzen
    frame->SetOcApp(OcApp);

    // Erstellen einer Statuszeile
    TStatusBar* sb = new TStatusBar(frame, TGadget::Recessed);

    // Erstellen einer Dialogelementleiste
    TControlBar* cb = new TControlBar(frame);
    cb->Insert(*new TButtonGadget(CM_FILENEW, CM_FILENEW,
TButtonGadget::Command));
    cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN,
TButtonGadget::Command));
    cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE,
TButtonGadget::Command));
    cb->Insert(*new TButtonGadget(CM_FILESAVEAS, CM_FILESAVEAS,
TButtonGadget::Command));
    cb->Insert(*new TSeparatorGadget);
    cb->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE,
TButtonGadget::Command));
    cb->Insert(*new TButtonGadget(CM_PENCOLOR, CM_PENCOLOR,
TButtonGadget::Command));
```

```
    cb->Insert(*new TSeparatorGadget);
    cb->Insert(*new TButtonGadget(CM_ABOUT, CM_ABOUT,
TButtonGadget::Command));
    cb->SetHintMode(TGadgetWindow::EnterHints);

    // Setzen des Bezeichners der Dialogelementleiste
    // (wird für die OLE-Mauspalettenänderung gebraucht)
    cb->Attr.Id = IDW_TOOLBAR;

    // Einfügen der Statuszeile und der Dialogelementleiste in den Rahmen
    frame->Insert(*sb, TDecoratedFrame::Bottom);
    frame->Insert(*cb, TDecoratedFrame::Top);

    // Erstellen des Hauptfensters und seines Menüs
    SetMainWindow(frame);
    GetMainWindow()->SetMenuDescr(TMenuDescr("MDI_COMMANDS",
                                  1, 1, 0, 0, 1, 1));

    // Installieren der Dokumentverwaltung
    SetDocManager(new TDocManager(dmMDI | dmMenu));
}
```

Ändern der Dok/View-Klassen

Für die Unterstützung von OLE-Containern sind einige Änderungen an Ihren Klassen *TDrawDocument* und *TDrawView* vorzunehmen. Für Ihre Dokumentklasse müssen Sie folgendes machen:

- Hinzufügen weiterer Informationen zur Registriertabelle für die Erzeugung von *TDrawDocument*-Dokument-Templates
- Ändern der Basisklasse in *TOleDocument*
- Modifizieren des Konstruktors
- Hinufügen zweier weiterer Funktionen, nämlich *GetLine* und *IsOpen*
- Modifizieren der Dateizugriffsfunktionen zum Laden und Speichern von OLE-Objekten

Die View-Klasse benötigt:

- Ändern der Basisklasse in *TOleView*
- Entfernen des *DragDC*-Elementes
- Modifizieren des Konstruktors und Dekonstruktors, um die *DragDC* betreffenden Anweisungen zu entfernen
- Modifizieren der Funktion *Paint* dahingehend, daß sie die *Paint*-Funktion der Basisklasse aufruft
- Modifizieren der Mausbefehle zwecks Überprüfung, ob der Benutzer ein eingebettetes OLE-Objekt auswählt

Diese Änderungen sind in den folgenden Abschnitten beschrieben.

Ändern der Dokumentregistrierung

Die von Ihnen weiter vorne erzeugte Registriertabelle beinhaltet die für die Erzeugung des Basisdokument-Template nötigen Informationen. Dies funktioniert hervorragend, solange allein die Dokumentverwaltung das Dokument-Template verwendet. Aber dafür, wie ObjectComponents die Dok/View-Klassen benutzt, sind einige weitere Informationen vonnöten:

- Einen Bezeichner-String. Für diesen Bezeichner verwenden Sie das REGDATA-Makro mit dem Schlüsselwert *progid*. Es handelt sich um einen dreiteiligen Bezeichner, dessen Teile durch Punkte getrennte textuelle Beschreibungen sind. Es dürfen keine Leer oder nicht-alphabetischen Zeichen (außer den Punkt-Trennzeichen) enthalten sein.
  - Der erste Teil des Bezeichners beschreibt die Gesamtanwendung. In unserem Beispiel-Code ist dieser erste Teil also *DrawPad*.
  - Der zweite Teil beschreibt den Teil der Anwendung, der in dem mit der Registriertabelle assoziierten Modul enthalten ist. Für die Anwendungsregistriertabelle aus dem Beispiel-Code ist dieser Teil des Bezeichners also *Application* und bei der Dokument-Registriertabelle *Document*.
  - Der dritte Teil muß eine Zahl sein, im Beispiel-Code die Zahl 1. Unterstützt Ihre Anwendung verschiedenen Dokumenttypen, so verwenden Sie jeweils eine andere Zahl.

Beachten Sie, daß der Benutzer Ihrer Anwendung dies nicht zu sehen bekommt. Es wird lediglich in der OLE-Registrierungsdatenbank des Systems eingetragen und sollte für die jeweilige Anwendung eindeutig sein.

- Eine Beschreibung der Dokumentklasse, wofür Sie das Makro REGDATA mit dem Schlüsselwert *discription* verwenden. Diese Beschreibung ist für den Benutzer Ihrer Anwendung gedacht. Der String erscheint in der OLE-Registrierungsdatenbank, wenn jemand ein Objekt in seinen Container einfügt.
- Eine Liste der Datentypen, die die Container-Anwendung an die Zwischenablage übergeben kann. Zur Registrierung von Zwischenablage-Formaten verwenden Sie das Makro REGFORMAT, welches fünf Parameter entgegennimmt:
  - Formatpriorität. Je kleiner der Wert ist, umso höher ist die Priorität. 0 gibt an, daß das Format von höchster Priorität ist. Versucht ein Benutzer, Daten in Ihre Anwendung einzufügen, so versucht die Zwischenablage dies mit der höchsten Priorität, die mit dem Format der Daten in der Zwischenablage verträglich ist, auszuführen.
  - Datenformat.
  - Darstellungsart der anzuzeigenden Daten. So kann beispielsweise ein Bitmap als Bitmap, inform formatierter Informationen (wie

Größe und Anzahl der Farben) oder als Hexdump angezeigt werden, und ein Objekt kann auch in Symbolform dargestellt werden.

- Wie die Daten übertragen werden, falls nichts anderes spezifiziert wurde. So mag beispielsweise der Benutzer, wenn das Übertragen als durch Ziehen und Ablegen spezifiert ist, die Daten lieber inform einer temporären Datei an den Container übergeben wollen.
- Ob das Dokument diesen Datentyp sowohl liefern wie auch empfangen kann.

Jede OLE-Anwendung *muß* spezifizieren, daß sie die Formate *ocrEmbedSource* und *ocrMetafilePict* handhaben kann. Standardmäßig registriert ObjectComponents immer *ocrLinkSource*. Sie werden aber im allgemeinen selbst *ocrLinkSource* registrieren lassen, um die Priorität niedriger setzen zu können. Zusätzlich können Sie *ocrBitmap* und *ocrDIB* registrieren lassen. Bedenken Sie, daß diese Formate die Datentypen angeben, die Ihre Anwendung an die Zwischenablage übertragen kann, also nicht die Datentypen, die Ihre Anwendung entgegennimmt. Das Einfügen der Daten in die Zwischenablage wird von ObjectComponents behandelt. Die genaue Bedeutung eines jeden dieser Werte ist im *ObjectWindows Referenzhandbuch* Kapitel 6 beschrieben.

Die folgende Registriertabelle zeigt, wie die Ihrige aussehen müßte. Die Werte für das REGFORMAT-Makro sind in Schritt 10 beschrieben.

```
BEGIN_REGISTRATION(DocReg)
    REGDATA(progid, "DrawContainer")
    REGDATA(description, "OWL Drawing Pad 2.5 Document")
    REGDATA(extension, "PTS")
    REGDATA(docfilter, "*.pts")
    REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt
| dtRegisterExt)
    REGFORMAT(0, ocrEmbedSource, ocrContent, ocrIStorage, ocrGet)
    REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict, ocrGet)
    REGFORMAT(2, ocrBitmap, ocrContent, ocrGDI|ocrStaticMed, ocrGet)
    REGFORMAT(3, ocrDib, ocrContent, ocrHGlobal|ocrStaticMed, ocrGet)
    REGFORMAT(4, ocrLinkSource, ocrContent, ocrIStream, ocrGet)
END_REGISTRATION
```

Ändern von TDrawDocument zwecks Handhabung eingebetteter OLE-Objekte

Sie müssen für die Unterstützung eingebetteter OLE-Objekte an *TDrawDocument* ein paar Änderungen vornehmen. Diese – ziemlich einfachen – Änderungen betreffen im wesentlichen das Lesen und Schreiben von OLE-Objekte enthaltenden Dokumenten. Die neue Basisklasse *TOleDocument* bietet die meisten der für die Handhabung eingebetteter OLE-Objekte benötigten Fähigkeiten. Hier nun eine Zusammenfassung der zu machenden Änderungen:

- Ändern der Basisklasse von *TDrawDocument* in *TOleDocument*

- Modifizieren des *TDrawDocument*-Konstruktors zwecks besserer Leistungsfähigkeit
- Entfernen der Funktion *IsOpen*
- Hinzufügen einiger Funktionsaufrufe in den Funktionen *Commit* und *Open*; letztere lesen und schreiben OLE-Objekte beinhaltende Dokumente

Ändern der Basisklasse von *TDrawDocument* in *TOleDocument*

Um Ihre Dokumentklasse für die Arbeit in einer ObjectComponents-Umgebung parat zu machen, müssen Sie die Basisklasse von *TFileDokument* in *TOleDocument* abändern. *TOleDocument* basiert auf der Klasse *TStorageDocument*, welche widerum auf *TDocument* basiert. Mit *TStorageDocument* lassen sich Verbunddokumente verwalten und speichern. Verbunddokumente ermöglichen es, mannigfaltige Dokumente in einer einzigen Datei abzuspeichern, ohne sich darüber den Kopf zerbrechen zu müssen, wie die einzelnen Objekte rausgeschrieben bzw. eingelesen werden. Über die Fähigkeiten von *TStorageDocument* hinaus bietet *TOleDocument* die Kommunikation mit einem OLE-Objekt, die Kontrolle und das Anzeigen eines OLE-Objektes sowie das Lesen und Schreiben des Objektes aus dem bzw. in den Speicher.

Um Ihre Basisklasse *TFileDocument* in *TOleDocument* abzuändern, müssen Sie zunächst alle Bezüge von *TFileDocument* in *TOleDocument* ändern. Dies ist kein Problem, da lediglich der eigentliche Name zu ändern ist; alle Funktionsköpfe, einschließlich der Konstruktoren der Basisklasse, bleiben gleich.

Erzeugen und Entfernen von *TDrawDocument*

Die einzige, am Konstruktor von *TDrawDocument* nötige Änderung dient (im Gegensatz zur Änderung der Basisklasse in *TOleDocument*) hauptsächlich der Steigerung der Leistungsfähigkeit der Zeichenbrett-Anwendung – hat also nichts mit der OLE-Funktionalität zu tun.

Entfernen Sie das Element *Lines* aus der Initialisierungsliste des Konstruktors.

Initialisieren Sie *Lines* im Rumpf des Konstruktors mit der initialen Größe 100, der Untergrenze 0 und einem Delta von 5.

Ihr Konstruktor hat nun etwa folgende Form:

```
TDrawDocument(TDocument* parent)
:
    TOleDocument(parent), UndoLine(0), UndoState(UndoNone)
{
    Lines = new TLines(100, 0, 5);
}
```

Am Destruktor brauchen Sie nichts zu ändern.

Entfernen der Funktion *IsOpen*

Die Funktion *IsOpen* ist aus der Klasse *TDrawDocument* zu entfernen, da sie durch die am Konstruktor gemachte Änderung überflüssig geworden

ist. Die Funktion überprüft nämlich die Gültigkeit des *Lines*-Elementes, aber *Lines* zeigt nun immer auf ein gültiges Objekt.

TStorageDocument bietet eine *IsOpen*-Funktion zur Überprüfung, ob das Dokumentobjekt ein gültiges Element *IStorage* hat. *IStorage* ist ein OLE 2-Konstukt für das Abspeichern und Wiederholen von Verbunddateien. Eine Verbunddatei ist gewissermaßen eine Datei mit Verweisen auf Objekte, die sich an verschiedenen anderen Orten befinden. Dem Benutzer erscheint die Verbunddatei als ein einziges Dokument. In Wirklichkeit aber sind die verschiedenen Elemente der Datei in diversen vom System bestimmten Bereichen abgespeichert und werden vom *IStorage*-Objekt verwaltet. Bei der Konstruktion eines Containers lassen Sie sich darauf ein, in Ihrer Anwendung Verbunddokumente zu unterstützen. Da diese Unterstützung allerdings von OLE-fähigen ObjectComponents-Klassen bereitgestellt wird, brauchen Sie sich nicht selbst um die Verwaltung der Verbunddokumente zu kümmern.

Zusammen mit dem Entfernen der Deklaration und Definition von *IsOpen* müssen Sie auch alle Zugriffe auf die *IsOpen*-Funktion eliminieren. Die Funktion wird aber nur einmal, und zwar in der Funktion *GetLine*, aufgerufen. Dort können Sie einfach die gesamte, den Aufruf enthaltende Anweisung löschen. Diese Anweisung überprüft die Gültigkeit des *TLine*-Objektes des Dokumentes, auf das das Datenelement *Lines* zeigt. Die Überprüfung ist jedoch nicht mehr nötig, da die von Ihnen am Konstruktor durchgeführte Änderung sicherstellt, daß jedes Dokumentobjekt immer mit einem gültigen *TLine*-Objekt assoziiert ist. Somit sollte Ihre *GetLine*-Funktion nun folgendermaßen aussehen:

```
TLine*
TDrawDocument::GetLine(uint index)
{
   return index < Lines->GetItemsInContainer() ? &(*Lines)[index] : 0;
}
```

Die Deklaration der *TDrawDocument*-Klasse müßte etwa wie folgt aussehen:

```
class _DOCVIEWCLASS TDrawDocument : public TOleDocument {
   public:
      enum {
         PrevProperty = TFileDocument::NextProperty-1,
         LineCount,
         Description,
         NextProperty,
      };
      enum {
         UndoNone,
         UndoDelete,
         UndoAppend,
```

```
        UndoModify
      };
      TDrawDocument(TDocument* parent = 0);
     ~TDrawDocument() { delete Lines; delete UndoLine; }

      // Implementierung virtueller Methoden von TDocument
      bool Open(int mode, const char far* path = 0);
      bool Close();
      bool Commit(bool force = false);
      bool Revert(bool clear = false);

      int FindProperty(const char far* name);   // Index zurückgeben
      int PropertyFlags(int index);
      const char far* PropertyName(int index);
      int PropertyCount() {return NextProperty - 1;}
      int GetProperty(int index, void far* dest, int textlen = 0);

      // Funktionen für den Datenzugriff
      TLine* GetLine(uint index);
      int AddLine(TLine& line);
      void DeleteLine(uint index);
      void ModifyLine(TLine& line, uint index);
      void Clear();
      void Undo();

    protected:
      TLines* Lines;
      TLine* UndoLine;
      int UndoState;
      int UndoIndex;
      string FileInfo;
};
```

Lesen und Schreiben eingebetteter OLE-Objekte

Die letzte Änderung machen Sie an Ihrer Dokumentklasse, um in einem Dokument eingebettete OLE-Objekte sichern sowie laden zu können. Die dazu nötigen Fähigkeiten finden Sie in zwei von *TOleDocument* gelieferten Funktionen namens *Open* und *Commit*. Wie Sie sich wahrscheinlich schon denken können, liest *Open* die in einem Dokument enthaltenen OLE-Objekte ein und *Commit* schreibt sie hinaus, führt also die Änderungen auf der Platte aus.

Für die Änderungen ist folgendes zu machen:

Fügen Sie in der Funktion *Commit* den Aufruf von *TOleView::Commit* unmittelbar vor der Erzeugung des *TOutStream*-Objektes (durch Aufruf der Funktion *OutStream*) ein.

Fügen Sie in der Funktion *Open* von *TDrawDocument* den Aufruf von *TOleView::Open* unmittelbar vor der Erzeugung des *TInStream*-Objektes (durch Aufruf der Funktion *InStream*) ein.

Das ist alles, was Sie zum Speichern von OLE-Objekten in Ihrem Dokument tun müssen. Ihre *Commit*-Funktion sieht nun etwa folgendermaßen aus:

```
bool TDrawDocument::Commit(bool force)
{
   TOleDocument::Commit(force);

   TOutStream* os = OutStream(ofWrite);

   if (!os)
      return false;

   // Zeilenzahl der Abbildung schreiben
   *os << Lines->GetItemsInContainer();

   // Beschreibung unter Verwendung eines Ressource-Strings anfügen
   *os << ' ' << FileInfo << '\n';

   // Iterator für das Array der Zeilen besorgen
   TLinesIterator i(*Lines);

   // solange der Iterator gültig ist (wir also noch Zeilen haben)
   while (i)
      // Kopieren der aktuellen Zeile (gegeben durch den Iterator) und
inkrementieren des Arrays
      *os << i++;

   delete os;
   SetDirty(false);
   return true;
}
```

Ihre *Read*-Funktion müßte von folgender Form sein:

```
bool TDrawDocument::Open(int mode, const char far* path)
{
   char fileinfo[100];

   TOleDocument::Open(mode, path);
   if (GetDocPath()) {
      TInStream* is = (TInStream*)InStream(ofRead);

      if (!is)
         return false;

      unsigned numLines;
      *is >> numLines;
      is->getline(fileinfo, sizeof(fileinfo));

      while (numLines--) {
```

```
            TLine line;
            *is >> line;
            Lines->Add(line);
         }

         delete is;

         FileInfo = fileinfo;
      }
      else {
         FileInfo = string(*::Module, IDS_FILEINFO);
      }
      SetDirty(false);
      UndoState = UndoNone;
      return true;
   }
```

Ändern von** TDrawView **zwecks Handhabung eingebetteter OLE-Objekte

An *TDrawView* müssen Sie zwecks Unterstützung eingebetteter OLE-Objekte ein paar Änderungen machen. Diese betreffen hauptsächlich die Handhabung von OLE-Objekten mit Hilfe der Maus, einschließlich Ziehens von Objekten und Aktivierung des Objekt-Servers. Die Änderungen sind ziemlich einfach: Die meisten zur Handhabung eingebetteter OLE-Objekte benötigten Fähigkeiten werden von der neuen Basisklasse *TOleView* geboten. Nachfolgend sehen Sie eine Übersicht über die nötigen Änderungen.

- Ändern Sie die Basisklasse von *TDrawView* in *TOleView*.
- Entfernen Sie das *DragDC*-Element; *TOleView* liefert einen *TDC*-Zeiger namens *DragDC*.
- Modifizieren Sie den Konstruktor und Dekonstruktor dahingehend, daß die Initialisierung und das Löschen von *DragDC* entfällt.
- Entfernen Sie die Funktion *EvRButtonDown*.
- Modifizieren Sie die *Paint*-Funktion so, daß diese *TOleView::Paint* aufruft, also die eingebetteten Objekte gezwungen sind, ihr eigenes *Paint* zu verwenden.
- Modifizieren Sie die Mausfunktionen dahingehend, daß diese bei eingebetteten OLE-Objekten mit der Interaktion mit dem Benutzer klarkommen.
- Modifizieren Sie die Klassendeklaration so, daß sie die Änderungen in der View-Klasse widerspiegelt.

Modifizieren der *TDrawView*-Deklaration

Hier sehen Sie die Klassendeklaration für *TDrawView*. Die Änderungen an ihr werden in den nachfolgenden Abschnitten erklärt.

```
class _DOCVIEWCLASS TDrawView : public TOleView {
   public:
```

```
      TDrawView(TDrawDocument& doc, TWindow* parent = 0);
     ~TDrawView() {delete Line;}
      static const char far* StaticName() {return "Draw View";}
      const char far* GetViewName() {return StaticName();}

   protected:
      TDrawDocument* DrawDoc;   // dasselbe wie das Doc-Element, aber
Zugriff auf die abgeleitete Klasse
      TPen* Pen;
      TLine* Line;   // zur Aufnahme einer einzelnen vom Dokument
geschickten oder empfangenen Zeile

      // Botschaftsantwortfunktionen
      void EvLButtonDown(uint, TPoint&);
      void EvMouseMove(uint, TPoint&);
      void EvLButtonUp(uint, TPoint&);
      void Paint(TDC&, bool, TRect&);
      void CmPenSize();
      void CmPenColor();
      void CmClear();
      void CmUndo();

      // Dokumentmeldungen
      bool VnCommit(bool force);
      bool VnRevert(bool clear);
      bool VnAppend(uint index);
      bool VnDelete(uint index);
      bool VnModify(uint index);

   DECLARE_RESPONSE_TABLE(TDrawView);
};
```

Und hier ist die Antworttabelle für *TDrawView*.

```
DEFINE_RESPONSE_TABLE1(TDrawView, TOleView)
   EV_WM_LBUTTONDOWN,
   EV_WM_MOUSEMOVE,
   EV_WM_LBUTTONUP,
   EV_COMMAND(CM_PENSIZE, CmPenSize),
   EV_COMMAND(CM_PENCOLOR, CmPenColor),
   EV_COMMAND(CM_EDITCLEAR, CmClear),
   EV_COMMAND(CM_EDITUNDO, CmUndo),
   EV_VN_COMMIT,
   EV_VN_REVERT,
   EV_VN_DRAWAPPEND,
   EV_VN_DRAWDELETE,
   EV_VN_DRAWMODIFY,
END_RESPONSE_TABLE;
```

Ändern der Basisklasse von *TDrawView* in *TOleView*

Damit Ihre View-Klasse in einer ObjectComponents-Umgebung arbeiten kann, müssen Sie die Basisklasse von *TWindowView* in *TOleView* abändern. *TOleView* basiert bereits selbst auf der *TWindowView*-Klasse und bietet die zur Manipulation und zum Verschieben von OLE-Objekten sowie die zur Aktivierung eines Servers des Objektes benötigten Fähigkeiten.

Um Ihre Basisklasse von *TWindowView* in *TOleView* abzuändern, müssen Sie erst aus allen Zugriffe auf *TWindowView* solche auf *TOleView* machen. Das ist recht einfach, da lediglich die eigentlichen Namen zu ändern sind; alle Funktionsköpfe, einschließlich derer der Basisklassen-Konstruktoren, bleiben gleich.

Entfernen von *DragDC*

Diese Änderung ist ziemlich offensichtlich. *TOleView* liefert einen Zeiger namens *DragDC* auf ein *TDC*, wodurch es einer solchen Elementes in der *TDrawView*-Klasse nicht länger bedarf. Somit müssen Sie auch diverse Arbeiten im Zusammenhang mit *DragDC* wieder herausnehmen. Um viele der Aufgaben, wie etwa das Erzeugen eines Gerätekontextobjektes, sobald die linke Maustaste betätigt wird, kümmert sich *TOleView*. Die Änderungen werden im nächsten Abschnitt, wo sie zum Tragen kommen, diskutiert.

Erstellen und Entfernen von *TDrawView*

Die einzige, am Konstruktor von *TDrawView* nötige Änderung ist das Entfernen der Initialisierung des *DragDC*-Elementes. Obwohl *DragDC* aus der Klassendeklaration von *TDrawView* gelöscht wurde, ist es immer noch ein Klassenelement; es wird von *TOleView* geliefert. Aber *TOleView* kümmert sich auch um die Initialisierung von *DragDC*, da *TOleView* überprüfen muß, ob der Benutzer OLE-Aktionen durchgeführt hat.

Beachten Sie, daß der *TOleView*-Konstruktorkopf der gleiche wie der von *TWindowView* ist, d.h. Sie brauchen außer dem Namen nichts zu ändern. Ihr *TDrawView*-Konstruktor sollte dann wie folgt aussehen:

```
TDrawView::TDrawView(TDrawDocument& doc, TWindow* parent) :
   TOleView(doc, parent), DrawDoc(&doc)
{
   Line = new TLine(TColor::Black, 1);
   SetViewMenu(new TMenuDescr(IDM_DRAWVIEW));
}
```

Entsprechend ist die einzige, am Destruktor von *TDrawView* nötige Modifikation das Entfernen der Anweisung, welche *DragDC* löscht.

```
~TDrawView()
{
   delete Line;
}
```

Modifizieren der *Paint*-Funktion

Die Funktion *Paint* ist dahingehend zu ändern, daß sie *TOleView::Paint* aufruft. *TOleView::Paint* findet jedes mit dem Dokument verknüpfte bzw. darin eingebettete Objekt (sofern vorhanden) und instruiert diese jeweils,

daß sie ihr eigenes *Paint* verwenden sollen. Ist dies geschehen, so können Sie ganz wie in Schritt 13 auf dem Bildschirm malen. Ihre neue *Paint*-Funktion sollte folgendermaßen aussehen:

```
void TDrawView::Paint(TDC& dc, bool erase, TRect&rect)
{
   TOleView::Paint(dc, erase, rect);

   // Iteration über das Array von Zeilenobjekten
   int j = 0;
   TLine* line;
   while ((line = const_cast<TLine *>(DrawDoc->GetLine(j++))) != 0)
      line->Draw(dc);
}
```

Auswahl von OLE-Objekten

Die nächsten durchzuführenden Änderungen betreffen Funktionen, die sich um Arbeiten mit der Maus kümmern, nämlich *EvLButton*, *EvMouseMove* und *EvLButtonUp*. In diesen Funktionen ist zu überprüfen, ob der Benutzer mit der Maus an einem OLE-Objekt arbeitet und welcher Zeichenmodus gesetzt ist. Dies wird weitgehend von *TOleView* erledigt; Sie brauchen dafür lediglich die Basisklassen-Versionen der Funktionen aufzurufen. Die Änderungen der einzelnen Funktionen werden in den folgenden Abschnitten diskutiert.

Modifizieren von *EvLButton*

An der grundlegenden Funktionsweise von *EvLButton* , so wie sie aus Schritt 13 vorliegt, brauchen Sie nichts zu ändern. Sie müssen aber Einiges hinzunehmen, um zu berücksichtigen, daß OLE-Objekte im View sein können.

Als erstes müssen Sie von der *TOleView*-Basisklasse überprüfen lassen, ob der Benutzer ein OLE-Objekt ausgewählt hat; dazu ist *TOleView::EvLButtonDown* aufzurufen. Diese Funktion deaktiviert ein gegebenenfalls aktuell selektiertes OLE-Objekt, erzeugt ein neues *TOleDC*-Objekt und prüft, ob ein anderes OLE-Objekt selektiert war. (*TOleDC* ist von der von Ihnen in vorangegangenen Schritten verwendeten Klasse *TClientDC* abgeleitet und ergänzt die Fähigkeit zur Handhabung eingebetteter OLE-Objekte.)

Zur Überprüfung, ob der Benutzer im View zeichnen will oder kann, ist zweierlei zu tun: Erstens ist festzustellen, ob mit Hilfe eines *TOleView::EvLButtonDown*-Aufrufs ein gültiger Gerätekontext erzeugt wurde, und zweitens, ob ein OLE-Objekt selektiert worden ist. Die Gültigkeit des Gerätekontextes läßt sich einfach über *DragDC* testen. Ob ein OLE-Objekt selektiert wurde, können Sie durch Aufruf der Funktion *SelectEmbedded* herausfinden. *SelectEmbedded* gibt **true** zurück, wenn ein Objekt selektiert worden ist, und sonst **false**. Falls beides nicht zutrifft, kann *EvLButtonDown* einfach zurückspringen.

Angenommen, es gibt einen gültigen Gerätekontext, aber es ist kein OLE-Objekt selektiert; dann können Sie ganz wie in Schritt 13 mit der Zeichenoperation beginnen. Sie müssen lediglich die Initialisierung von *DragDC* löschen, da *DragDC* bereits auf einen gültigen Gerätekontext gesetzt wurde.

Ihre *EvLButtonDown*-Funktion sieht nun etwa wie folgt aus:

```
void TDrawView::EvLButtonDown(uint modKeys, TPoint& point)
{
   TOleView::EvLButtonDown(modKeys, point);

   if (DragDC && !SelectEmbedded()) {
      SetCapture();
      Pen = new TPen(Line->QueryColor(), Line->QueryPenSize());
      DragDC->SelectObject(*Pen);
      DragDC->MoveTo(point);
      Line->Add(point);
   }
}
```

Modifizieren von *EvMouseMove*

Die für *EvMouseMove* nötigen Änderungen sind ähnlich denen in *EvLButtonDown*.

Rufen Sie die Version von *EvMouseMove* aus der Basisklasse auf.

Überprüfen Sie, ob der Gerätekontext gültig und eine OLE-Objekt selektiert ist.

Setzen Sie die Zeichenoperation ganz wie in Schritt 13 fort.

Dann sieht ihre *EvMouseMove*-Funktion ungefähr wie folgt aus:

```
void TDrawView::EvMouseMove(uint modKeys, TPoint& point)
{
   TOleView::EvMouseMove(modKeys, point);

   if (DragDC && !SelectEmbedded()) {
      DragDC->LineTo(point);
      Line->Add(point);
   }
}
```

Modifizieren von *EvLButtonUp*

Mit *EvLButtonUp* müssen Sie das gleiche machen, was Sie schon mit *EvLButtonDown* und *EvMouseMove* gemacht haben, allerdings etwas andersherum. In diesem Fall ist die Basisklassen-Version der Funktion als letztes anstatt als erstes aufzurufen. *TOleView::EvLButtonUp* führt einige Aufräumoperationen aus und löscht dabei auch das Gerätekontextobjekt, auf das *DragDC* zeigt.

Überprüfen Sie, ob der Gerätekontext gültig und eine OLE-Objekt selektiert worden ist.

Führen Sie die gleichen Operationen wie in Schritt 13 aus, allerdings ohne das Löschen und ohne *DragDC* auf 0 zu setzen.

Rufen Sie *TOleView::EvLButtonUp* auf.

Ihre *EvLButtonUp*-Funktion hat nun folgende Form:

```
void TDrawView::EvLButtonUp(uint modKeys, TPoint& point)
{
   if (DragDC && !SelectEmbedded()) {
      ReleaseCapture();
      if (Line->GetItemsInContainer() > 1) {
         DrawDoc->AddLine(*Line);
      }
      Line->Flush();
      delete Pen;
   }

   TOleView::EvLButtonUp(modKeys, point);
}
```

Schritt 14: Erstellen eines OLE-Servers

Daß Ihre Anwendung OLE-Server unterstützt, indem sie sich als OLE-Container darstellt, ist im Hinblick auf Flexibilität ein großer Schritt vorwärts für Ihre Anwendung. Ihre Funktionalität erweitert sich damit in jede nur denkbare Richtung. Aber eines fehlt noch: Könnten Sie aus Ihrer Anwendung einen OLE-Server machen, so könnte Ihre Anwendung die Funktionalität anderer Anwendungen erweitern.

Stellen Sie sich etwa vor, Sie entwerfen Datenbankformulare und wollen diese mit einigen Ihrer Abbildungen etwas attraktiver gestalten. Ohne OLE ist das Einbinden Ihrer Abbildungen reichlich beschwerlich, da Sie jede Abbildung irgendwie erfassen und in Ihr Formular einsetzen müssen. Und ist sie dann erst in Ihrem Formular, können Sie sie nicht mehr ändern, außer Sie bedienen sich wieder Ihrer Zeichenbrettanwendung, um die Abbildung dann erneut einbauen zu müssen.

Ist die Datenbank hingegen ein OLE-Container, und haben Sie aus Ihrer Zeichenbrettanwendung einen OLE-Server gemacht, dann können Sie Abbildungen ohne weiteres in den Datenbankformularen ablegen. Der eingebettete OLE-Server hilft Ihnen gegebenenfalls beim Modifizieren der Abbildungen, ohne daß Sie die Datenbankanwendung verlassen müssen.

Dieser Schritt beschreibt, wie Sie aus Ihrer Dok/View-Zeichenbrettanwendung aus Schritt 12 einen OLE-Server machen. Den Code dieses Beispiels finden Sie in den Dateien, step15dv.cpp, step15.cpp, STEP15.H, STEP15DV.H, step15.rc und step15dv.rc im Verzeichnis EXAMPLES\OWL\TUTORIAL Ihres installierten Compilers.

Hinweis

Nach Ausführen der Änderungen dieses Schrittes wird Ihre Zeichenbrettanwendung ausschließlich eine Server-Anwendung sein, d.h. sie kann nicht länger eingebettete OLE-Objekte beinhalten. Damit soll die der Anwendung beigegebene spezielle Server-Funktionalität demonstriert werden. Die Änderungen, die die Unterstützung der Container herausnehmen, werden als solche angegeben. Möchten Sie also die Container und Server-Unterstützung in einer einzigen Anwendung kombinieren, müssen Sie lediglich diese entsprechenden Änderungen dann auslassen.

Konvertieren Ihres Anwendungsobjekts

Damit Ihr Anwendungsobjekt zu einem OLE-Server wird, müssen Sie ein paar Änderungen vornehmen:

- Ändern der Header-Dateien
- Ändern der Registriertabelle der Anwendung
- Ändern des Basisklassenkonstruktors, um einige weitere Informationen zu speichern (einschließlich des Anwendungs-Dictionary)
- Verbergen des Fensters, wenn die Anwendung als Server aktiviert wurde
- Hinzufügen von Parametern für Modulbezeichner zu einigen Objektkonstruktoren
- Erzeugung neuer Views ändern
- Auffinden des zum Info-Dialogfenster übergeordneten Fensters ändern
- Ändern der Funktion *OwlMain*, um nach Bearbeitungsoptionen zu suchen

Ändern der Header-Dateien

In der Liste der Header-Dateien in step15.cpp brauchen Sie lediglich zwei Änderungen durchführen.

- Fügen Sie die Header-Datei owl\oleview.h hinzu; die Klasse *TOleView* wird zum Erzeugen neuer Views benötigt.
- Binden Sie step15.rc statt step14.rc ein.

Ändern der Registriertabelle der Anwendung

Grundsätzlich müssen Sie die gesamte Registriertabelle aus Schritt 13 abändern. Allerdings hängt nur eine dieser Änderungen unmittelbar mit der Umwandlung der Anwendung in einen OLE-Server zusammen.Die mit den Schlüsseln *clsid* und *description* assoziierten Werte sind zu ändern, da es sich um eine von Schritt 13 verschiedene Anwendung handelt.

Ihre Registriertabelle sollte nun wie folgt aussehen:

```
BEGIN_REGISTRATION(AppReg)
   REGDATA(clsid, "{5E4BD320-8ABC-101B-A23B-CE4E85D07ED2}")
   REGDATA(description, "OWL Drawing Pad Server")
END_REGISTRATION
```

Hinweis

Vergessen Sie nicht: Sie dürfen den GUID und den Programmbezeichner für andere Anwendungen nicht einfach übernehmen! Um durch dergleichen entstehende Duplizitäten zu verhindern, wurden alle Werte von Schritt 13 in Schritt 14 geändert.

Ändern des Konstruktors der Anwendung

Für OLE-Server muß der Basisklassenkonstruktor von *TDrawApp* dahingehend geändert werden, daß er das Objekt für das Anwendungs-Dictionary als Parameter übernimmt. Für Container-Anwendungen war dies nicht nötig. Der Grund dafür ist, daß Container immer als ausführbare Anwendung erzeugt werden (im Gegensatz zu einer DLL). Geben Sie im Konstruktor von *TApplication* kein Anwendungs-Dictionary an, so wird das globale Anwendungs-Dictionary *::OwlAppDictionary* verwendet. Bei ausführbaren Anwendungen funktioniert das: Da diese eine eigene Instanz besitzt, wird sie in das globale Anwendungs-Dictionary eingetragen. DDLs hingegen haben keine eigene Instanz. Anwendungsobjekte müssen ihr eigenes Anwendungs-Dictionary auch selbst verwalten.

TApplication bietet einige Parameter mehr, als Sie bisher benutzt haben. Der erste ist der Name der Anwendung, den Sie im vorangegangenen Schritt zum Setzen des Anwendungsnamens verwendet haben.

Der zweite Parameter ist eine Referenz eines Zeigers auf ein Objekt *TModule* (also *Tmodule*&). Der Konstruktor von *TApplication* setzt diesen Zeiger so, daß er auf das neue Anwendungsobjekt zeigt. In diesem Fall wollen Sie das globale Modulobjekt *::Module* übergeben; *::Module* wird von ObjectWindows und ObjectComponents zur Identifizierung des aktuellen Moduls verwendet. Beachten Sie, daß *::Module* der Vorgabewert für diesen Parameter ist.

Der letzte Parameter ist ein Zeiger auf ein Objekt *TAppDictionary*. Verwenden Sie einen Zeiger auf das Objekt *TAppDictionary*, den Sie sich mit Hilfe des Makros DEFINE_APP_DICTIONARY besorgen.

Dann sieht Ihr Konstruktor folgendermaßen aus:

```
TDrawApp() : TApplication(::AppReg["description"], ::Module,
                          &::AppDictionary) {}
```

Verbergen des Hauptfensters des Servers

Unter normalen Umständen führt Ihre Anwendung beim Starten gewisse Initialisierungen durch und erzeugt dann ein Hauptfenster für die Arbeiten des Benutzers. Das ist auch völlig in Ordnung, wenn der Benutzer Ihre Anwendung als seinen primären Arbeitsplatz verwendet. Wird Ihre Anwendung hingegen als OLE-Server benutzt, ist sie eben *kein* primärer Arbeitsplatz; das Hauptfenster ist dann bereits von einer anderen Anwendung erzeugt worden. In diesem Fall müssen Sie also dafür sorgen, daß Ihr Hauptfenster verborgen bleibt.

Dies läßt sich am besten in *InitMainWindow* unmittelbar vor der Erzeugung des Fensterobjekts erreichen. Zunächst ist festzustellen, ob die Anwendung als eingebetteter Server arbeitet, um dann gegebenenfalls das Hauptfenster zu verbergen:

1. Rufen Sie die Funktion *IsOptionSet* des Objekts *TOcRegistrar* auf, und übergeben Sie *TOcCmdLine::Embedding* als Funktionsargument. Eine Referenz auf das Registrarobjekt der Anwendung erhalten Sie durch Aufruf der Funktion *GetRegistrar*. Die Funktion *IsOptionSet* überprüft die Kommandozeile der Anwendung darauf, ob die ihr als Parameter übergebene Option darin enthalten ist. Wurde eine Anwendung als eingebetteter Server erzeugt, so ist die Option *Embedding* in der Kommandozeile angegeben. Folglich gibt *IsOptionSet* **true** zurück, wenn ihr *TOcCmdLine::Embedding* als Parameter übergeben wurde und die Anwendung als eingebetteter Server erzeugt worden ist. Weitere dieser Optionen werden wir später noch kennenlernen.
2. Gibt *IsOptionSet* **true** zurück, d.h. die Anwendung wird als eingebetteter Server aufgerufen, so setzen Sie *nCmdShow* auf SW_HIDE. Dies bewirkt, daß das Hauptfenster bei seiner Erzeugung verborgen und die Aktivierung an das Fenster weitergereicht wird, das den Server aktiviert hat.

Identifizieren des Moduls

Bei der Erstellung der Basisklasse *TApplication* hatten Sie einige neue Parameter hinzuzufügen, um sicherzustellen, daß das Objekt sich selbst in der verwickelten OLE-Umgebung findet. Dasselbe müssen Sie für einige andere Objekte Ihrer Anwendung machen. Im Fall dieser Objekte brauchen Sie diese allerdings nur an das Anwendungsobjekt zu verweisen, welches alle Transaktionen zwischen Ihrer Anwendung und dem, was sich außerhalb von dieser befindet, handhabt.

- Das MDI-Client-Fenster braucht einen einzigen Paramter (einen Modul-Zeiger).

- Der OLE-MDI-Rahmen braucht einen Zeiger *TModule* ale Parameter nach dem Parameter für das Nachführen des Menüs (der der zuletzt verwendete Parameter in Schritt 13 war)
- Die Dokumentverwaltung braucht nach seinem Flags-Parameter einen Zeiger *TApplication*.

Verwenden Sie den Zeiger **this** von *TDrawApp* für alle diese Parameter.

Ihre Funktion *InitMainWindow* sollte nun von folgender Form sein:

```
void
TDrawApp::InitMainWindow()
{
   if (GetRegistrar().IsOptionSet(TOcCmdLine::Embedding))
      nCmdShow = SW_HIDE;

   TOleMDIFrame* frame;
   frame = new TOleMDIFrame(GetName(), 0, *(Client = new
TMDIClient(this)), true, this);
   frame->SetOcApp(OcApp);

   // Erstellen einer Statuszeile
   TStatusBar* sb = new TStatusBar(frame, TGadget::Recessed);

   // Erstellen einer Mauspalette
   TControlBar* cb = new TControlBar(frame);
   cb->Insert(*new TButtonGadget(CM_FILENEW, CM_FILENEW,
TButtonGadget::Command));
   cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN,
TButtonGadget::Command));
   cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE,
TButtonGadget::Command));
   cb->Insert(*new TButtonGadget(CM_FILESAVEAS, CM_FILESAVEAS,
TButtonGadget::Command));
   cb->Insert(*new TSeparatorGadget);
   cb->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE,
TButtonGadget::Command));
   cb->Insert(*new TButtonGadget(CM_PENCOLOR, CM_PENCOLOR,
TButtonGadget::Command));
   cb->Insert(*new TSeparatorGadget);
   cb->Insert(*new TButtonGadget(CM_ABOUT, CM_ABOUT,
TButtonGadget::Command));
   cb->SetHintMode(TGadgetWindow::EnterHints);
   cb->Attr.Id = IDW_TOOLBAR;

   // Einfügen der Statuszeile und der Mauspalette in den Rahmen
   frame->Insert(*sb, TDecoratedFrame::Bottom);
   frame->Insert(*cb, TDecoratedFrame::Top);

   // Erstellen des Hauptfensters und seiner Menüs
```

```
    SetMainWindow(frame);
    GetMainWindow()->SetMenuDescr(TMenuDescr(IDM_MDICMNDS));

    // Installieren der Dokumentverwaltung
    SetDocManager(new TDocManager(dmMDI | dmMenu, this));
}
```

Erzeugen neuer Views

Beim Erzeugen eines neuen View-Fensters in der OLE-Server-Anwendung müssen Sie vorsichtig sein, wenn Sie das Fenster bestimmen, das dem View übergeordnetet ist. Wenn Ihre Anwendung als eigenständiges Programm läuft, brauchen Sie nichts zu ändern; der in den vorangegangenen Schritten verwendete Code von *EvNewView* ist genau richtig.

Schwieriger wird es, wenn der Server in einer Container-Anwendung eingebettet ist. Sie müssen dann etwas ganz Entscheidendes feststellen, nämlich ob Ihr View Platz in einem der Fenster des Containers benötigt. Dazu müssen Sie die beiden folgenden Fragen beantworten:

- Wird die Anwendung als eingebetteter Server verwendet? Lautet die Antwort nein (d.h. Ihre Anwendung läuft eigenständig), können Sie die nächste Frage überspringen. Sie wissen dann, daß Ihre Anwendung keinen Platz im Fenster des Containers benötigt, weil es nämlich keinen Container gibt.
- Wurde die Anwendung zwecks Editierens geöffnet? Der Benutzer kann auf Ihren eingebetteten Server in zweierlei Form zugreifen: Entweder durch In-place-Editieren – dann sitzt der Arbeitsbereich Ihres Servers im Arbeitsbereich des Containers – oder durch offenes Editieren – dann öffnet sich der Server für das Editieren und sieht etwa genauso aus, als arbeite er selbständig. Öffnet der Benutzer den Server zum Editieren, dann nutzt der Server *keinen* Platz gemeinsam mit dem Container-Fenster. Nur wenn der Benutzer Ihren Server zum Editieren vor Ort verwendet, müssen Sie sich um die gemeinsame Bereichsnutzung mit dem Container kümmern.

Dies muß bestimmt werden, da davon das Setzen des übergeordneten Fensters für das View abhängt. Wird der Server als In-place-Server betrieben, so müssen Sie das übergeordnete Fenster entsprechend setzen, wofür ObjectComponents zur Vereinfachung ein als View-Eimer bekanntes Objekt zur Verfügung stellt. Sobald Sie das übergeordnete Fenster Ihres Views auf den View-Eimer gesetzt haben, sorgt ObjectComponents dafür, daß das übergeordnete Fenster bei Aktionen wie Aktivieren, Deaktivieren, Verschieben usw. richtig gesetzt wird. Um das dem View übergeordnete Fenster zu setzen, gehen Sie also wie folgt vor:

1. Wandeln Sie den Parameter *TView* der Funktion *EcNewView* in *TOleView* um. Holen Sie die Adresse des Objekts (durch Voranstellen eines kauf-

männischen Unds &), und weisen Sie sie mit Hilfe des Makros TYPESAFE_DOWNCAST einem Zeiger *TOleView* zu.

2. Prüfen Sie durch Aufruf der Funktion *IsEmbedded* aus dem mit dem View assoziierten Dokument, ob das View ein eingebetteter Server ist. Das View selbst weiß nicht, ob es eingebettet ist. Das mit dem View assoziierte Dokument finden Sie durch Aufruf der Funktion *GetDocument* des Views. Ist das Dokument nicht eingebettet, so können Sie die Überprüfung hiermit beenden und zu dem in den vorangegangenen Schritten verwendeten Code übergehen.
3. Stellen Sie fest, ob das View für offenes Editieren aktiviert wurde, indem Sie die Funktion *IsOpenEditing* des Remote Views dieses Views aufrufen. Einen Zeiger auf das Remote View erhalten Sie nach Aufruf von *GetOcRemView*. Gibt *IsOpenEditing* **true** zurück, dann können Sie die Überprüfung hier abbrechen und zu dem in den vorangegangenen Schritten verwendeten Code übergehen.
4. Nachdem Sie nun herausbekommen haben, daß die Anwendung als Server für ein direktes Bearbeiten verwendet wird, können Sie mit dem Setzen des übergeordneten Fensters für das View fortfahren. Gehen Sie wie folgt vor:

1. Bestimmen Sie das mit dem View assoziierte Fenster. Sie erhalten mit Hilfe der Funktion *GetWindow* des Views einen Zeiger *TWindow* auf dieses Fenster.
2. Suchen Sie den mit dem Server assoziierten Remote View-Eimer. Rufen Sie dazu die Funktion *GetMainWindow* auf und wandeln Sie den Rückgabewert in einen Zeiger *TOleFrame* um. *TOleFrame* bietet eine Funktion namens *GetRemViewBucket*, welche einen auf den Remote View-Eimer verweisenden Zeiger *TWindow* zurückgibt.
3. Nachdem Sie nun den Remote View-Eimer gefunden haben, rufen Sie die Funktion *SetParent* des Views mit dem auf den Eimer verweisenden Zeiger *TWindow* auf.
4. Rufen Sie die Funktion *Create* des Views auf.

Beachten Sie, daß Sie eigentlich nicht das übergeordnete Fenster des Views so gesetzt haben, wie Sie es sich normalerweise vorstellen. Aber durch den Remote-View-Eimer brauchen Sie das nur einmal zu tun, und dann kümmert sich ObjectComponents darum, immer das aktuelle übergeordnete Fenster zu verfolgen.

Der Code für diese Funktion sieht etwa folgendermaßen aus:

```
void
TDrawApp::EvNewView(TView& view)
{
```

```
      TOleView* ov = TYPESAFE_DOWNCAST(&view, TOleView);
      if (view.GetDocument().IsEmbedded() &&
!ov->GetOcRemView()->IsOpenEditing()) {
         TWindow* vw = view.GetWindow();
         vw->SetParent(TYPESAFE_DOWNCAST(GetMainWindow(),
TOleFrame)->GetRemViewBucket());
         vw->Create();
      }
      else {
         TMDIChild* child = new TMDIChild(*Client, 0);
         if (view.GetViewMenu())
            child->SetMenuDescr(*view.GetViewMenu());
         child->Create();
         child->SetClientWindow(view.GetWindow());
      }
   }
```

Ändern des übergeordneten Fensters des Info-Dialogfensters

In vorangegangenen Versionen der Beispielanwendung haben Sie beim Erzeugen des Info-Dialogfensters einfach die Funktion *GetMainWindow* aufgerufen, um das dem Info-Dialogfenster übergeordnete Fenster zu finden. Dies geht nun nicht mehr, da Sie nicht wissen, ob Ihr Hauptfenster wirklich das ist, was der Benutzer als Hauptfenster sieht. Ist Ihre Anwendung in einer anderen Anwendung eingebettet, so haben Sie bereits im Konstruktor *TDrawApp* festgestellt, daß sie Ihr Hauptfenster nicht anzeigen.

Um das fokussierte Fenster oder irgendein anderes passendes View-Fenster (das als übergeordnetes Fenster dient) auf der Arbeitsfläche zu finden, können Sie die Funktion *GetCommandTarget* aufrufen. Diese Funktion wird von *TFrameWindow* bereitgestellt und gibt ein Handle auf das aktuell aktive Fenster zurück. Beachten Sie, daß der Aufruf dieser Funktion unabhängig davon funktioniert, ob die Anwendung als eingebetteter Server läuft oder als eigenständige Anwendung, da einfach der Befehl Fokussiertes Fenster zurückgegeben wird. Ist die Beispielanwendung ein eingebetteter Server, ist das zurückgegebene Handle eines auf das fokussierte Fenster der Client-Anwendung. Läuft die Beispielanwendung hingegen eigenständig, so wird ein Handle auf sie selbst geliefert.

Beachten Sie, daß Sie trotzdem *GetMainWindow* aufrufen müssen, um einen Zeiger auf das Hauptfenster der Beispielanwendung zu bekommen. Dann rufen Sie die Funktion *GetCommandTarget* dieses Fensterobjekts auf. Außerdem müssen Sie ein temporäres *TWindow* erzeugen, um den Rückgabewert von *GetCommandTarget* an den Konstruktor *TDialog* zu übergeben. Ihre modifizierte Funktion *CmAbout* sieht dann wie folgt aus:

```
void
TDrawApp::CmAbout()
{
   TDialog(&TWindow(GetMainWindow()->GetCommandTarget()),
```

```
IDD_ABOUT).Execute();
  }
```

Modifizieren von *OwlMain*

Sie müssen sich lediglich um eines kümmern, bevor Sie eine OLE-Server-Anwendung laufen lassen: Sie müssen die Kommandozeile daraufhin überprüfen, ob eine der standardmäßigen Aktionsoptionen spezifiziert wurde.

Es gibt mehrere standardmäßige ObjectComponents-Kommandozeilenoptionen, die bei einer Server-Anwendung spezifiziert werden können. Das Vorliegen einer dieser Aktionsoptionen bedeutet, daß Ihre Anwendung nicht wie gewöhnlich auszuführen ist, sondern eine bestimmte Aufgabe durchführen soll, um dann wieder beendet zu werden. Bei einer OLE-Server-Anwendung müssen Sie auf folgende Aktionsoptionen achten:

- Die Option *RegServer* teilt Ihrer Anwendung mit, daß sie sich vollständig in der OLE-Registrierungsdatenbank eintragen soll.
- Die Option *UnregServer* teilt Ihrer Anwendung mit, daß sie ihren Eintrag aus der OLE-Registrierungsdatenbank entfernen soll.

Das Gute an diesen Optionen ist, daß ObjectComponents diese Aktionen automatisch für Sie ausführt, wenn Sie das Registrarobjekt erzeugen. Das einzige, was Sie tun müssen, ist, in der Funktion *OwlMain* zu überprüfen, ob eine dieser Optionen gesetzt war. Falls ja, können Sie sofort zurückspringen; falls nein, fahren Sie mit dem nächsten Schritt fort.

Sie können diese Optionen mit Hilfe der Funktion *IsOptionSet* überprüfen, die Sie auch in der Funktion *InitMainWindow* für die Untersuchung des Flags *Embedding* verwendet haben. Für die hiesigen Optionen müssen Sie das Flag *TOcCmdLine::AnyRegOptions* untersuchen. Dieses Flag überprüft, ob eine der für Ihre Anwendung relevanten Optionen gesetzt wurde; *IsOptionSet* gibt **true** zurück, wenn das der Fall ist.

War eines der Flags gesetzt, so können Sie in *OwlMain* 0 zurückgeben. ObjectComponents führt die Registrier-Task durch, wenn eine dieser Aktionsoptionen gesetzt ist. Ist dies geschehen, so ist die Anwendung fertig. Ihre Anwendung geht also nach einer Registrier-Task niemals zur normalen Ausführung über.

Ihre Funktion *OwlMain* müßte nun folgendermaßen aussehen:

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
   Registrar = new TOcRegistrar(AppReg, TOleFactory<TDrawApp>(),
                                TApplication::GetCmdLine());

   if (Registrar->IsOptionSet(TOcCmdLine::RegServer |
```

```
    TOcCmdLine::UnregServer))
            return 0;

        return Registrar->Run();
    }
```

Ändern Ihrer Dok/View-Klassen

Es gibt einige Änderungen, die Sie zwecks Unterstützung der OLE-Server-Funktionalität an Ihren Dok/View-Klassen vornehmen müssen:

- Ändern Sie die Header-Dateien.
- Modifizieren Sie die Registriertabelle, um zusätzliche, vom OLE-Server benötigte Informationen, zu liefern.
- Ändern Sie die View-Meldungsfunktionen *VnRevert*, *VnAppend*, *VnModify* und *VnDelete*.
- Fügen Sie einige neue Elemente – einschließlich eines Zeigers *TControlBar* und einiger neuer Funktionen – zu *TDrawView* hinzu.
- Löschen Sie Funktionsaufrufe aus den Funktionen für die Mausaktionen und der Funktion *Paint*.

Ändern der Header-Dateien

Sie müssen Ihre Liste von Header-Dateien ändern, indem Sie einige neue Header-Dateien einbinden und die Ressourcen-Script-Datei für Schritt 14 hinzunehmen. Die neu einzubindenden Dateien sind owl\controlb.h und owl\buttonga.h. Ihre Include-Anweisungen sollten dann etwa folgende sein:

```
#include <owl\c.h>
#include <owl\inputdia.h>
#include <owl\chooseco.h>
#include <owl\gdiobjec.h>
#include <owl\docmanag.h>
#include <owl\listbox.h>
#include <owl\controlb.h>
#include <owl\buttonga.h>
#include <owl\olemdifr.h>
#include <owl\oledoc.h>
#include <owl\oleview.h>
#include <classlib\arrays.h>
#include "step15dv.rc"
```

Ändern der Dokumentregistriertabelle

Um ein OLE-Server sein zu können, müssen Sie an Ihrer Dokumentregistriertabelle einige recht umfangreiche Änderungen vornehmen. Die Teile, die sich nicht ändern, werden in diesem Abschnitt betrachtet.

Das Definieren der Registriertabelle unterscheidet sich nicht gegenüber vorher, also insbesondere was die Benutzung der Makros BEGIN_REGISTRATION und END_REGISTRATION betrifft. Ganz wie zuvor beginnt Ihre Tabelle mit dem Makro BEGIN_REGISTRATION, das als einzigen Parameter den Namen der Registrierung entgegennimmt. Das Makro END_REGISTRATION beendet die Tabellendefinition.

Die beiden Makros REGDATA, die die Tabelleneinträge *extension* und *docfilter* setzen, bleiben auch gleich, ebenso wie das Makro REGDOCFLAGS.

Die zu ändernden Teile der Registriertabelle werden in den nächsten Abschnitten diskutiert.

Programmbezeichner und -beschreibung

Der Programmbezeichner (der mit dem Schlüssel *progid* assoziierte Wert) und die Beschreibung (der mit dem Schlüssel *description* assoziierte Wert) aus Schritt 13 beschreiben die Anwendung als »Draw Container« bzw. »OWL Drawing Pad Container«. Diese Werte müssen dahingehend geändert werden, daß sie die Tatsache wiedergeben, daß es sich bei der Anwendung um einen Server handelt.

Einfügen der Anwendung ermöglichen

ObjectComponents bietet einen speziellen Schlüssel namens *insertable*. Sie können *insertable* unter Verwendung des Makros REGDATA anmelden. Der mit dem Schlüssel *insertable* assoziierte Wert ist irrelevant; er wird nie benutzt, so daß sie etwa einen leeren String für ihn verwenden können.

Das Vorhandensein des Schlüssels *insertable* teilt ObjectComponents mit, daß die Anwendung einfügbar ist, also in andere Anwendungen eingebettet werden kann. Alle ObjectComponents-Server müssen den Schlüssel *insertable* in ihrer Registriertabelle spezifizieren!

Setzen der Menüeinträge des Servers

Aktiviert der Benutzer einen Server, der in einen Container ist, indem er das View des Servers anklickt, so setzt der Container einen Menüeintrag (gewöhnlich im Menü Bearbeiten), den der Benutzer für den Zugriff auf den Server verwenden kann. Dieses Menü wird zu einem Popup-Menü, das einige »Aktionsarten« zur Auswahl bietet, also Menüeinträge, mittels derer der Benutzer mit der Server-Anwendung arbeiten und die in ihr enthaltenen Daten manipulieren kann.

Folglich müssen Sie dafür zweierlei bereitstellen:

- Sie müssen den Menünamen bezeichnen, den der Container in seinem Menü Bearbeiten zur Repräsentation Ihrer Anwendung benutzen soll. Dazu können Sie das Makro REGDATA mit dem Schlüssel *menuname* und dem für das Menü gewünschten Text als Schlüsselwert verwenden. Sie sollten bei der Auswahl des Namens Rücksicht auf die Contai-

ner-Anwendung nehmen: Wählen Sie einen Namen, den Sie auch sonst in einem Menü verwenden würden, also einen, der beschreibend für Ihre Anwendung, aber nicht so lang ist, daß er das Menü durch seinen Eintrag unnötig aufbläht. In unserem Fall kann der Anwendungsname etwa »Zeichenbrett« sein.

- Sie können (mit Hilfe des Makros REGDATA) bis zu 20 Aktionsarten für Ihre Server-Anwendung spezifizieren. Die für REGDATA nötigen Schlüsselwerte sind gemäß dem Format *verb*n anzugeben, wobei n eine Zahl zwischen 0 und 19 bezeichnet. Der Wert, den Sie mit dem jeweiligen *verb* (der Aktionsart) assoziieren, ist der im Popup-Menü erscheinende Text. Beachten Sie, daß Sie für jede Aktionsart ein Tastenkürzel vergeben können, indem Sie dem Buchstaben des Tastenkürzels ein kaufmännisches Und (&) voranstellen. Spezifizieren Sie also beispielsweise Editieren als Aktionsart und möchten, daß der Benutzer durch Betätigen der Taste E dies aktivieren kann, so geben Sie den String »&Editieren« als Wert an.

Beachten Sie, daß die erste Aktionsart in der Liste (also die mit Schlüssel *verb0* assoziierte) die vorgegebene Aktionsart für Ihren Server ist. Wenn also der Benutzer Ihren eingebetteten Server doppelt anklickt, so reagiert der Server genauso, als hätte er den Wert *verb0* aus dem Server-Menü selektiert.

ObjectComponents-Server werden so eingerichtet, daß sie automatisch zwei Aktionsarten behandeln können.

- Die Editier-Aktionsart gibt an, daß der Benutzer die vom Server im Container gehandhabten Daten manipulieren möchte; d.h. der Benutzer arbeitet mit den Daten unmittelbar im Remote View-Bereich im Fenster des Containers.
- Die Aktionsart *Öffnen* gibt an, daß der Benutzer eine Server-Anwendung öffnen möchte, um die Daten zu manipulieren. In diesem Fall startet die Anwendung, als wäre sie selbst vom Benutzer aufgerufen worden. Der Hauptunterschied zwischen der Verwendung des Servers in dieser Form und der, ihn als eigenständige Anwendung laufen zu lassen, besteht darin, daß der Server in eine vom Container zur Verfügung gestellte Dokumentdatei schreibt; der Containerspeicher für Verbunddokumente kümmert sich dann um die Einzelheiten des Speicherns der Daten auf Platte.

Spezifizieren von Zwischenablageformaten

Sie können für die Server-Anwendung die Anzahl der verfügbaren Zwischenablageformate herabsetzen. Sie brauchen wirklich nur zwei Formate anzugeben:

- *ocrEmbedSource* gibt an, daß der Server als einbettbarer Quelltext in die Zwischenablage kopiert werden kann. Versucht jemand, einen einbett-

baren Quelltext von der Zwischenablage einzufügen, so landet eine Kopie des eingebetteten Server-Objekts in seiner Anwendung.

- *ocrMetafilePict* gibt an, daß der Server in der Darstellung *Metadatei* in die Zwischenablage kopiert werden kann.

Wie gehabt wird die eigentliche Kopieroperation von ObjectComponents ausgeführt. Beachten Sie, daß dies die einzigen zur Unterstützung eines ObjectComponents-Servers nötigen Formate sind; die übrigen, von der Container-Anwendung gelieferten Formate, werden gelöscht. Wünschen Sie, sowohl die Container als auch die Server-Funktionalität zu unterstützen, so sollten Sie die Formate bestehen lassen.

Ihre endgültige Dokumentregistriertabelle sollte dann von folgender Form sein:

```
    BEGIN_REGISTRATION(DocReg)
       REGDATA(progid, "DrawServer")
       REGDATA(menuname, "Drawing Pad")
       REGDATA(description, "OWL Drawing Pad Server")
       REGDATA(extension, "PTS")
       REGDATA(docfilter, "*.pts")
       REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir |
dtCreatePrompt | dtRegisterExt)
       REGDATA(insertable, "")
       REGDATA(verb0, "&Edit")
       REGDATA(verb1, "&Open")
       REGFORMAT(0, ocrEmbedSource, ocrContent, ocrIStorage, ocrGet)
       REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict, ocrGet)
    END_REGISTRATION
```

Ändern der View-Meldungsfunktionen

Die View-Meldungsfunktionen bedürfen einiger Änderungen zur Unterstützung einer korrekten Darstellung im Remote View des Servers. Die zu ändernden Funktionen sind *VnRevert, VnAppend, VnModify* und *VnDelete*. Jede dieser View-Meldungen gibt an, daß die Abbildung irgendwie geändert worden und die Darstellung auf den neuesten Stand zu bringen ist.

Um den Container zu zwingen, das View zu aktualisieren und somit die gemachten Änderungen in die Darstellung zu übernehmen, müssen Sie die Funktion *InvalidatePart* aufrufen. Diese Funktion wird von der Basisklasse *TDrawView TOleView* bereitgestellt. Sie teilt dem Container-Fenster mit, daß der Bereich im Remote View des eingebetteten Servers ungültig ist und somit eines neuen Anzeigens bedarf. *InvalidatePart* nimmt einen Parameter entgegen, nämlich ein *TOcInvalidate* **enum**. Dieses *TOcInvalidate* kann einer der beiden folgenden Werte sein:

- *invData* gibt an, daß die Daten im eingebetteten Objekt geändert wurden und im Container aktualisiert werden sollten.

- *invView* gibt an, daß sich die Erscheinungsform eines Objekts geändert hat und im Container aktualisiert werden sollte.

In unserem Fall geben beide View-Meldungsereignisse an, daß die Abbildung geändert wurde, egal ob es sich um die Rücknahme von Änderungen, das Hinzufügen einer weiteren Linie oder das Ändern oder Löschen einer Linie handelt. Wenn Sie die Funktion *InvalidatePart* aufrufen, so sollten Sie dies mit dem Argument *invView* machen. Das Argument *invData* wird benutzt, wenn der Container eine Verbindung mit den Daten im Server hat, aber tatsächlich sich selbst um die Anzeige der Daten kümmert.

Sie sollten zunächst die Funktion *Invalidate* des Views aufrufen, d.h. falls anwendbar. (Jede der Funktionen außer *VnAppend*, die dies nicht braucht, ruft bereits *Invalidate* auf.) Dann sollten Sie die Funktion *InvalidatePart* aufrufen. Ihre modifizieren View-Meldungsfunktionen sind dann etwa von folgender Form:

```
bool
TDrawView::VnRevert(bool /*clear*/)
{
   Invalidate();   // gesamte Neuausgabe erzwingen
   InvalidatePart(invView);
   return true;
}

bool
TDrawView::VnAppend(uint)
{
   InvalidatePart(invView);
   return true;
}

bool
TDrawView::VnModify(uint /*index*/)
{
   Invalidate();   // gesamte Neuausgabe erzwingen
   InvalidatePart(invView);
   return true;
}

bool
TDrawView::VnDelete(uint /*index*/)
{
   Invalidate();   // gesamte Neuausgabe erzwingen
   InvalidatePart(invView);
   return true;
}
```

Hinzufügen neuer Elemente zu *TDrawView*

Sie müssen der Klasse *TDrawView* einige neue Elemente hinzufügen, und zwar

- einen Zeiger *TControlBar*
- zwei neue Ereignisbehandlungsroutinen zum Ausschneiden und Kopieren
- zwei neue Ereignisbehandlungsroutinen für ObjectComponents-Ereignisse

Hinzufügen einer Mauspalette

Wenn Ihre Anwendung als eingebetteter Server aktiviert wird, erwartet der Container häufig die Bereitstellung einer Mauspalette, um auf ihre Funktionalität zugreifen zu können. Diese Mauspalette sollte sich von der der normalen Anwendung unterscheiden und nur Schalter-Gadgets für die speziellen Funktionen der Anwendung, nicht aber solche für die von den Containern gehandhabten Funktionen (also das Editieren von Objekten und die View-Befehle) bereitstellen. Beispielsweise wird das Öffnen einer Datei von der passenden Container-Anwendung gehandhabt, ist also keine spezielle Fähigkeit der Zeichenbrettanwendung. Andererseits weiß aber kein Container, wie die Stiftfarbe der Zeichenbrettanwendung zu ändern ist.

Da die von dieser Mauspalette unterstützten Befehle eine echte Teilmenge der von der Mauspalette der Anwendung unterstützten Befehle ist, können Sie nicht einfach diese verwenden. Statt dessen müssen Sie für jedes Server/View eine eigene Mauspalette bereitstellen. Zu diesem Zweck fügen Sie einfach einen Zeiger *TControlBar* als geschütztes Datenelement hinzu, welches Sie im Konstruktor von *TDrawView* mit 0 initialisieren. Die eigentliche Mauspalette wird in einem der neuen ObjectComponents-Ereignisbehandlungsroutinen erstellt (siehe die nächsten Quelltexte).

Ausschneiden und Kopieren von Daten

An Ihren Server werden häufig Anfragen gestellt werden, Daten auszuschneiden und/oder in die Zwischenablage zu kopieren. Zur Bearbeitung dieser Anfragen müssen Sie Funktionen bereitstellen.

Ausschneiden

Ausschneiden bedeutet, daß die Informationen aus der Abbildung kopiert, in die Zwischenablage übernommen und dann aus der Abbildung entfernt werden. Dies ist eine übliche Methode, um Daten auszutauschen. Allerdings ist die Wirkung im Zusammenhang mit der Zeichenbrettanwendung nicht definiert: Was bedeutet es, Linien aus einem Fenster auszuschneiden?

Da es sich jedoch um eine in OLE-Servern sehr verbreitete (nahezu obligatorische) Funktion handelt, sollten Sie zumindest einen Platzhalter dafür liefern. Sie können zu diesem Zweck eine Funktion namens *CmEditCut* deklarieren und definieren. Diese Funktion wird aufgerufen, sobald *TDrawView* ein Ereignis CM_EDITCUT empfängt; auch dieses müssen Sie

einbauen (befindet sich in der Datei step15dv.rc im Beispiel-Code). Gehen Sie also wie folgt vor:

1. Fügen Sie das Makro CM_EDITCUT zu Ihrer Anwendung hinzu.
2. Ergänzen Sie die Klassendeklaration *TDrawView* um die Funktion *CmEditCut*.
3. Fügen Sie zur Antworttabelle ein Makro EV_COMMAND hinzu, das bei Empfang des Ereignisses CM_EDITCUT *CmEditCut* aufruft.
4. Definieren Sie *CmEditCut* als wirkungslos.

Kopieren

Zum Kopieren können Sie eine der von ObjectComponents bereitgestellten Elementfunktionen aufrufen. Die Klasse heißt *TOcRemView* und liefert ein Remote View-Objekt für eine Server-Dokument. Ein Remote View bearbeitet das View Ihrer Server-Anwendung von der Container-Anwendung aus. *TOcRemView* bietet eine Funktion namens *Copy*, die die Daten des Dokuments in die Zwischenablage kopiert. Sie können das Objekt *TOcRemView* durch Aufruf der von *TOleView* gelieferten Funktion *GetOcRemView* zur Mitarbeit veranlassen.

1. Fügen Sie Ihrer Anwendung das Makro CM_EDITCOPY hinzu.
2. Erweitern Sie die Klassendeklaration *TDrawView* um die Funktion *CmEditCopy*.
3. Fügen Sie zur Antworttabelle das Makro EV_COMMAND hinzu, um *CmEditCopy* aufzurufen, sobald das Ereignis CM_EDITCOPY empfangen wird.
4. Definieren Sie *CmEditCopy* so, daß *GetOcRemView* und dann die Funktion *Copy* des Objekts *TOcRemView* aufgerufen wird.

Behandeln von ObjectComponents-Ereignissen

Es gibt ein paar ObjectComponents-Ereignisse, auf die Sie reagieren müssen.

- OC_VIEWPARTSIZE weißt auf eine Anfrage von seiten des Containers bezüglich der Größe des View-Objekts hin, d.h. also die Größe des »Fensters« innerhalb des Container-Fensters, in dem der Beutzer Ihre eingebettete Anwendung sieht.
- OC_VIEWSHOWTOOLS gibt an, daß der Container eine Anfrage bezüglich der Mauspalette der Server-Anwendung gestellt hat.

Melden der Größe des Server Views

Aus Formatierungsgründen muß der Container häufig die Größe des Views des eingebetteten Servers ausfindig machen. Der Container signalisiert durch Senden einer ObjectComponents-Botschaft an das View, daß er diese Informationen braucht. Das View muß dann die Größe des Server-Views berechnen und die Informationen an den Container zurückgeben.

So fügen Sie diese Fähigkeit hinzu:

1. Der Container läßt den Server durch Senden von OC_VIEWPARTSIZE – einem standardmäßigen ObjectComponents-Ereignis – wissen, daß er die Größe des Views benötigt. ObjectWindows bietet dafür ein Antworttabellenmakro sowie weitere standardmäßige ObjectComponents-Ereignismakros. Die ObjectComponents-Ereignismakros sind in der Header-Datei owl\ocfevent.h definiert, welche automatisch eingebunden wird. Bei den Makros ist dem Namen des ObjectComponents-Ereignisses ein EV_ vorangestellt, d.h. in diesem Falle wäre EV_OC_VIEWPARTSIZE das Makro. Fügen Sie dieses der Antworttabelle Ihres Views hinzu. Wie andere standardmäßige Botschaftsmakros hat es keine Parameter und ruft beim Empfang des Ereignisses einen vordefinierten Funktionsnamen auf.
2. Erweiteren Sie Ihre Klassendeklaration *TDrawView* um eine Funktion zur Behandlung dieses Ereignisses. Die durch das vordefinierte Antworttabellenmakro aufgerufene Funktion ist *EvOcViewPartSize*; sie gibt einen Wert vom Typ **bool** zurück und nimmt einen Zeiger auf ein *TRect* entgegen.
3. Definieren Sie die Funktion *EvOcViewPartSize*. Dazu erzeugen Sie ein Gerätekontextobjekt (im hiesigen Beispiel-Code haben wir ein *TClientDC* verwendet). Die Größe des Views sollte dem Objekt *TRect*, das als Zeiger an *EvOcViewPartSize* übergebenen wird, zugewiesen werden. In der Zeichenbrettanwendung ist die View-Größe auf dem Bildschirm auf 5 cm beschränkt. Dies ist eine beliebige Größe; Sie können also ebensogut die von den Informationen Ihres Dokuments benötigte Größe ausrechnen und diese dann zurückgeben. Wir geben lediglich aus Gründen der Einfachheit einen festen Wert zurück. In diesem Fall sind die Elemente *top* und *left* von *TRect* auf 0 zu setzen. Sie können sich die Anzahl der Pixel für diese View-Größe dann durch Aufruf der Funktion *GetDeviceCaps* des Gerätekontextobjekts besorgen; der Parameter LOGPIXELSX liefert die Breite und LOGPIXELSY die Höhe. Tatsächlich wird die Anzahl der Pixel pro Zoll des Bildschirms zurückgegeben. Multiplizieren Sie das jeweilige Ergebnis mit zwei, und weisen Sie die Breite dem Element *right* und die Höhe dem Element *bottom* des Objekts*TRect* zu.

Die vollständige Funktion sieht dann wie folgt aus:

```
bool
TDrawView::EvOcViewPartSize(TRect far* size)
{
   TClientDC dc(*this);

   // 2" x 2" für den Server
   size->top    = size->left = 0;
```

```
    size->right  = dc.GetDeviceCaps(LOGPIXELSX) * 2;
    size->bottom = dc.GetDeviceCaps(LOGPIXELSY) * 2;
    return true;
}
```

Setzen der Mauspalette des Views

Das Ereignis OC_VIEWSHOWTOOLS besagt, daß der Container, in dem der Server eingebettet ist, die Mauspalette des Servers entweder anzeigen oder verbergen will. Dazu ist einiges zu tun:

1. Fügen Sie zur Antworttabelle Ihres Views das Makro OC_VIEWSHOWTOOLS hinzu.
2. Erweitern Sie Ihre Klassendeklaration *TDrawView* um eine Funktion zur Behandlung dieses Ereignisses. Die mittels des vordefinierten Antworttabellenmakros aufgerufene Funktion ist *EvOcViewShowTools*. Diese Funktion nimmt einen Verweis auf ein Objekt *TOcToolBarInfo* entgegen und gibt einen **bool**-Wert zurück. *TOcToolBarInfo* ist eine einfache Struktur; sie hat nur wenige Elemente, die uns hier betreffen.
   - Das erste ist das Element *Show* vom Typ **bool**. Ist *Show* **true**, so möchte der Container eine Mauspalette anzeigen, bei **false** die Mauspalette verbergen.
   - Das zweite Element ist *HTopTB*, ein HWND. Sie geben die Mauspalette über dieses Element an den Container zurück.
3. Will der Container die Mauspalette verbergen (wenn also *Show* **false** ist), so müssen Sie das Mauspalettenfenster entfernen, das Mauspalettenobjekt löschen und Ihren Zeiger *TControlBar* auf 0 setzen. Vorher sollten Sie sich jedoch vergewissern, daß der Zeiger *TControlBar* auf ein gültiges Objekt verweist.
4. Will der Container die Mauspalette anzeigen, so sollten Sie zunächst überprüfen, ob der Zeiger *TControlBar* nicht vielleicht bereits auf ein gültiges Objekt zeigt. Falls ja, können Sie den nächsten Punkt überspringen und mit dem übernächsten fortfahren.
5. Das Kompliziertesteste bei der Erstellung einer Mauspalette ist unter diesen Umständen die Bestimmung des übergeordneten Fensters. Dazu sind einige Schritte nötig, da Sie Ihr Hauptfenster finden müssen, um dann mittels dieses OLE-Rahmenfensters den Remote View-Eimer, den die Anwendung im Fenster des Containers verwendet, zu suchen.

1. Als erstes ist das Anwendungsobjekt zu finden. Dies ist nämlich der einfachste Weg zur Auffindung des Hauptfensters, da das Anwendungsobjekt eine Funktion bietet, die einen Zeiger auf das Hauptfenster besorgt. Verwenden Sie für die Suche nach dem Anwendungsobjekt die Funktion *GetApplication*; diese gibt einen Zeiger *TApllication* auf das Anwendungsobjekt zurück.

2. Sobald Sie das Anwendungsobjekt gefunden haben, können Sie sich durch Aufruf von dessen Funktion *GetMainWindow* einen Zeiger *TFrameWindow* auf das Hauptfenster besorgen.

3. Nachdem Sie nun das Hauptfenster haben, müssen Sie es in ein Fenster *TOleFrame* umwandeln, um das Fenster des Remote View-Eimers finden zu können. Obwohl das Hauptfenster bereits ein Objekt *TOleFrame* ist, gibt *GetMainWindow* es als *TFrameWindow* zurück. Da Sie ein Objekt der Basisklasse in ein Objekt einer daraus abgeleiteten Klasse umwandeln, müssen Sie vorsichtig sein. Es ist möglich, zu versuchen, ein Objekt eines Typs in ein Objekt eines anderen Typs umzuwandeln zu versuchen; sind aber beide Typen aus derselben Basisklasse abgeleitet, so kann das ernsthafte Probleme geben.

   Stellen Sie sich etwa eine Funktion vor, die als einzigen Parameter einen Zeiger *TWindow* entgegennimmt. Beim Aufruf der Funktion nehmen Sie an, daß der in der Funktion erhaltene Wert *TWindow* tatsächlich ein Objekt *TControl* bezeichnet. (Da *TControl* von *TWindow* abgeleitet ist, können Sie ein Objekt *TControl* ohne weiteres als Objekt *TWindow* übergeben.) Aber *TControl* und *TFrameWindow* sind beide von *TWindow* abgeleitet. Was passiert also, wenn das hereingegebene Objekt eigentlich ein *TFrameWindow*-Objekt war? Das kann zu ernsthafter Verwirrung führen.

   ObjectWindows bietet ein Makro namens TYPESAVE_DOWNCAST, das Objekte einer Basisklasse in solche eines abgeleiteten Typs umwandelt. Ist die Umwandlung nicht typsicher – wenn das Objekt also nicht das ist, worin Sie es umwandeln wollen, etwa im Falle eines in ein *TControl* umzuwandelnden *TFrameWindow* – , so gibt das Makro 0 zurück. Anderenfalls führt das Makro die Umwandlung durch und gibt den passenden Wert zurück.

   TYPESAVE_DOWNCAST braucht zwei Parameter: Der erste ist das umzuwandelnde Objekt und der zweite der Typ, in den Sie dieses Objekt umwandeln wollen.

4. Nachdem Sie das Hauptfenster der Anwendung gefunden und entsprechend umgewandelt haben, müssen Sie die Funktion *GetRemViewBucket* aufrufen. Diese Funktion liefert einen Zeiger *TWindow*, der auf das Fenster des Remote View-Eimers verweist. Dies ist ziemlich wichtig: Ist das der Mauspalette übergeordnete Fenster richtig gesetzt, so kann der Container leicht zwischen den Mauspaletten der gegebenenfalls in den Container eingebetteten Server hin- und herschalten.

5. Nachdem Sie nun einen Zeiger auf das Fenster des Remote View-Eimers erzeugt haben, konstruieren Sie das Objekt *TControlBar* wie gewöhnlich, indem Sie den View-Zeiger als übergeordnet übergeben.

Ist das Mauspalettenobjekt erstellt, so können Sie Schalter-Gadgets zur Kontrolle der Anwendung einfügen. Im Moment reicht es, die Schalter CM_PENSIZE und CM_PENCOLOR hinzuzufügen.

6. Sobald Sie ein gültiges Mauspaletten-Objekt haben, erzeugen Sie durch Aufruf der Funktion *Create* des Objekts die eigentliche Mauspalette.
7. Nach Erzeugen der Mauspalette wandeln Sie diese in ein HWND um und weisen Sie dem Element *HTopTB* von *TOcToolBarInfo* zu. Sie können es statt dessen auch einem der anderen Elemente von *TOcToolBarInfo* zuweisen, um es irgendwo anders als oben in das Fenster des Containers zu plazieren.
8. In der Annahme, daß alles gut funktioniert hat, geben Sie **true** zurück. Dadurch weiß der Container, daß alles geklappt hat und er die Mauspalette anzeigen kann.

   Ihre *EvOcViewShowTools*-Funktion müßte nun folgendermaßen aussehen:

```
bool
TDrawView::EvOcViewShowTools(TOcToolBarInfo far& tbi)
{
   // Erzeugen einer Mauspalette zum Anzeigen, Löschen zwecks
Verbergens
   if (tbi.Show) {
      if (!ToolBar) {
         TOleFrame* frame =
TYPESAFE_DOWNCAST(GetApplication()->GetMainWindow(), TOleFrame);
         ToolBar = new TControlBar(frame->GetRemViewBucket());
         ToolBar->Insert(*new TButtonGadget(CM_PENSIZE, CM_PENSIZE,
TButtonGadget::Command));
         ToolBar->Insert(*new TButtonGadget(CM_PENCOLOR, CM_PENCOLOR,
                         TButtonGadget::Command));
      }
      ToolBar->Create();
      tbi.HTopTB = (HWND)*ToolBar;
   }
   else {
      if (ToolBar) {
         ToolBar->Destroy();
         delete ToolBar;
         ToolBar = 0;
      }
   }
   return true;
}
```

Entfernen von Aufrufen aus der Funktion *Paint* und den Mausfunktionen

Die Funktion *Paint* von *TDrawView* sowie die Funktionen *EvLButtonDown*, *EvLButtonUp* und *EvMouseMove* für die Mausaktionen führen jeweils Aufrufe zur Unterstützung der Container-Funktionalität durch. Soll Ihre Anwendung ausschließlich als Server-Anwendung arbeiten, so sollten Sie diese Aufrufe entfernen.

- Die Funktion *Paint* ruft *TOleView::Paint* auf, damit alle eingebetteten Objekte aufgerufen werden und erfahren, daß sie sich selbst anzeigen sollen. Da eine ausschließlich als Server arbeitende Anwendung keine eingebetteten Objekte hat, ist dieser Aufruf unnötig.
- *EvLButtonDown*, *EvLButtonUp* und *EvMouseMove* rufen die Funktion *SelectEmbedded* auf, um festzustellen, ob der Benutzer ein eingebettetes Objekt angeklickt – und damit selektiert – hat. Hier gilt das gleiche wie bei *Paint*: Da es bei einer ausschließlich als Server arbeitenden Anwendung keine eingebetteten Objekte gibt, ist der Aufruf überflüssig.

K A P I T E L 3

Anwendungsobjekte

ObjectWindows 2.5 kapselt Windows-Anwendungen und DLL-Module, wobei auf die Klassen *TApplication bzw. TModule* zurückgegriffen wird. Objekte der Klasse *TModule* kapseln die Funktionen zum Initialisieren und Schließen einer DLL von Windows. Das *TModule*-Objekt beinhaltet ebenso die Parameter *hInstance* und *lpCmdLine*, die äquivalent zu den gleichnamigen Parametern sind, welche in einer Nicht-ObjectWindows-Anwendung an die *WinMain*-Funktion übergeben werden. Beachten Sie, daß sowohl *WinMain* als auch *LibMain* diese beiden Parameter gemeinsam haben. Die Klasse *TModule* wird unten in Kapitel 16 genauer beschrieben.

TApplication-Objekte kapseln Funktionen zum Initialisieren, zur Laufzeitverwaltung und zum Schließen einer Windows-Anwendung. Das *TApplication*-Objekt beinhaltet ebenso die Werte der Parameter *hPrevInstance* und *nCmdShow*, die äquivalent zu den gleichnamigen Parametern sind, welche in einer Nicht-ObjectWindows-Anwendung an die *WinMain*-Funktion übergeben werden. Da *TApplication* auf *TModule* basiert, besitzt *TApplication* auch die gesamte Funktionalität, die in *TModule* enthalten ist.

Zusätzlich enthält das *TApplication*-Objekt Funktionen, um die Bibliothek für Dialogelemente im Borland-Stil und die Bibliothek für 3D-Dialogelemente von Microsoft auf einfache Weise zu laden und zu benutzen. Ferner existiert eine spezielle Funktion, die Standarddialogelemente automatisch als Subklassen von 3D-Dialogelementen von Microsoft vorsieht. Siehe im folgenden für weitere Informationen.

Für Ihre ObjectWindows 2.5-Anwendung müssen Sie keine explizite *WinMain*-Funktion bereitstellen. Greifen Sie stattdessen auf die Funktion *OwlMain* zurück. *OwlMain* ermöglicht Ihnen die Verwendung der Parameter **int** argc und **char** ***argv* und gibt ein **int** zurück, wie es auch im traditionellen C- oder C++ Programm bei der *main*-Funktion üblich ist. Siehe weiter unten für weitere Informationen.

Dieses Kapitel beschreibt die Anwendung von *TApplication*-Objekten. Es sollte für Sie niemals notwendig sein, selbst ein *TModule*-Objekt zu erzeugen. Wenn Sie mit einer DLL arbeiten, sollten Sie die Klasse *TLibId* verwenden. Siehe weiter unten für weitere Informationen über den Gebrauch von DLLs in einer ObjectWindows-Anwendung.

Die Minimalanforderungen

Um ein *TApplication*-Objekt zu benutzen, müssen Sie zuerst:

- Die richtige Header-Datei miteinbinden
- Ein Objekt erzeugen
- Das Objekt finden

Miteinbinden der Header-Datei

TApplication ist in der Header-Datei owl\applicat.h definiert. Um *TApplication* verwenden zu können, müssen Sie diese Header-Datei miteinbinden. Da *TApplication* von *TModule* abgeleitet ist, bindet owl\applicat.h die Header-Datei owl\module.h mit ein.

Erzeugen eines Objektes

Sie können ein *TApplication*-Objekt erzeugen, indem Sie einen der beiden Konstruktoren benutzen. Der am häufigsten verwendete Konstruktor ist der folgende:

```
TApplication(const char far* name);
```

Diese Version des *TApplication*-Konstruktors erwartet einen String, der zum Namen der Anwendung wird. Wenn Sie keinen Namen angeben, wird die Anwendung vom Kontruktor standardmäßig mit dem leeren String benannt. *TApplication* verwendet diesen String als Anwendungsname.

Die zweite Version des *TApplication*-Konstruktors ermöglicht Ihnen die Angabe einer ganzen Zahl von Parametern, die mit den normalerweise an die *WinMain*-Funktion übergebenen Parametern korrespondieren:

```
TApplication(const char far* name,
             HINSTANCE instance,
             HINSTANCE prevInstance,
             const char far* cmdLine,
             int cmdShow);
```

Sie können diesen Konstruktor dazu verwenden, um Kommandoparameter an das *TApplication*-Objekt zu übergeben.

Finden des Objektes

TApplication beinhaltet mehrere Elementfunktionen und Datenelemente, die Sie für einen Aufruf Ihrer Anwendungsobjekte von außerhalb eventuell benötigen. Um einen Zugriff darauf zu ermöglichen, besitzt die Klasse *TWindow* eine Elementfunktion mit dem Namen *GetApplication*, die einen Zeiger auf das Anwendungsobjekt zurückgibt. Mit Hilfe dieses Zeigers können Sie dann Elementfunktionen von *TApplication* aufrufen und auf Datenelemente von *TApplication* zugreifen. Das unten stehende Beispiel zeigt eine mögliche Verwendung von *GetApplication*.

```
void TMyWindow::Error() {
     // zeige das Meldungsfenster an, das den Anwendungsnamen enthält
     MessageBox("An error occurred!",
                GetApplication()->Name, MB_OK);
     }
```

Erzeugen der minimalen Anwendung

Hier folgt die kleinste ObjectWindows-Anwendung, die Sie erzeugen können:

```
#include <owl\applicat.h>

int OwlMain(int argc,char* argv[])
{
  return TApplication("Wow!").Run();
}
```

Dies erzeugt eine Windows-Anwendung mit einem Hauptfenster mit Namen "Wow!". Sie können dieses Fenster in der Größe verändern, bewegen, als Symbol darstellen, auf volle Ausdehnung vergrößern und schließen. In einer echten Anwendung müßten Sie allerdings neue Klassen für die Anwendung ableiten, um die Funktionalität zu erweitern.

Initialisieren von Anwendungen

Für die Initialisierung einer ObjectWindows-Anwendung sind vier Schritte erforderlich:

- Konstruktion des Anwendungsobjektes
- Initialisierung der Anwendung
- Initialisierung jeder neuen Instanz
- Initialisierung des Hauptfensters

Konstruieren des Anwendungsobjektes

Wenn Sie ein *TApplication*-Objekt konstruieren, ruft dies seine Elementfunktionen aus *InitApplication*, *InitInstance* und *InitMainWindow* auf, um die Anwendung zu starten. Sie können jede dieser Funktionen überschrei-

ben, um die Initialisierung Ihrer Anwendung nach Ihren eigenen Bedürfnissen anzupassen. Sie müssen allerdings *InitMainWindow* überschreiben, um eine sinnvolle Anwendung zu erhalten. Um eine Funktion in *TApplication* zu überschreiben, müssen Sie Ihre eigene Anwendungsklasse von *TApplication* ableiten.

Der hier verwendete Konstruktor von *TApplication* benötigt als einziges Argument den Anwendungsnamen; sein Vorgabewert ist Null als Zeichen dafür, daß kein Name vorliegt. Auf den Anwendungsnamen wird für den Standardtitel des Hauptfensters und für Fehlermeldungen zurückgegriffen. Der Konstruktor Ihrer Anwendungsklasse sollte den Konstruktor von *TApplication* aufrufen. Das unten stehende Beispiel zeigt ein Fragment einer von *TApplication* abgeleiteten Klasse:

```
#include <owl\applicat.h>

class TMyApplication: public TApplication
{
public:
     TMyApplication(const char far* name = 0) : TApplication(name) {}

};
```

ObjectWindows 2.5-Anwendungen erfordern keine explizite *WinMain*-Funktion; die ObjectWindows-Bibliotheken stellen bereits eine Funktion zur Verfügung, die die Behandlung von Fehlern und Exceptions durchführt. Sie können jede gewünschte Initialisierung in der *OwlMain*-Funktion ausführen, die durch die *WinMain*-Standardfunktion aufgerufen wird.

Um ein Anwendungsobjekt zu konstruieren, müssen Sie eine Instanz Ihrer Anwendungsklasse in der *OwlMain*-Funktion erzeugen. Das unten stehende Beispiel zeigt die Definition und Instantiierung eines einfachen Anwendungsobjektes:

```
#include <owl\applicat.h>

class TMyApplication: public TApplication
{
public:
     TMyApplication(const char far* name = 0): TApplication(name) {}
     };

int OwlMain(int argc, char* argv[]) {
     return TMyApplication ("Wow!").Run();
     }
```

Verwenden von WinMain und OwlMain

ObjectWindows 2.5 enthält eine *WinMain*-Standardfunktion, die ausführliche Fehlertests und Exception-Behandlung bietet. Diese *WinMain*-Funktion baut die Anwendung auf und ruft die *OwlMain*-Funktion auf.

Auch wenn Sie Ihr eigenes *WinMain* verwenden können, indem Sie es einfach in eine Quelltextdatei plazieren, gibt es dafür wenig Anlaß. Alles, was Sie sonst in *WinMain* tun würden, können Sie auch in Initialisierungselementfunktionen von *OwlMain* oder *TApplication* tun. Das folgende Beispiel zeigt eine typische Verwendung von *OwlMain* in einer Anwendung:

```
#include <owl\applicat.h>
#include <string.h>

class TMyApplication: public TApplication
{
public:
     TMyApplication(const char far* name = 0): TApplication(name) {}
     };

int OwlMain(int argc, char* argv[])
{
     char title[30];

     if(argc = 2)
                strcpy(title;argv[1]);
     else
                strcpy(title;"Wow!");
     return TMyApplication(title).Run();
}
```

Wenn Sie entscheiden, Ihr eigenes *WinMain* zu verwenden, unterstützt *TApplication* die Übergabe von traditionellen *WinMain*-Funktionsparametern mit einem anderen Konstruktor. Das folgende Beispiel zeigt den Gebrauch dieses Konstruktors, um *WinMain*-Argumente an das *TApplication*-Objekt zu übergeben:

```
#include <owl\applicat.h>

class TMyApplication : public TApplication
{
public:
  TMyApplication (const char far* name,
                  HINSTANCE       instance,
                  HINSTANCE       prevInstance,
                  const char far* cmdLine,
                  int             cmdShow)
     : TApplication (name, instance, prevInstance, cmdLine, cmdShow);
     .
     .
     .
     };

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine, int nCmdShow)
{
     return TMyApplication("MyApp";hInstance;hPrevInstance,
```

```
                                    lpszCmdLine;nCmdShow).Run();
}
```

Initialisieren der Anwendung

Benutzer können mehrere Kopien einer Anwendung gleichzeitig laufen lassen. Aus der Sicht einer 16-Bit-Anwendung findet die Initialisierung der Erstinstanz nur dann statt, wenn eine andere Kopie der Anwendung nicht aktuell läuft. Die Initialisierung einer beliebigen Instanz findet jedesmal statt, wenn der Benutzer die Anwendung startet. Wenn ein Benutzer eine Anwendung startet, schließt, wieder startet usw. wird jede Instanz als Erstinstanz behandelt, da die Instanzen nicht zur gleichen Zeit laufen.

Im Fall von 32-Bit-Anwendungen läuft jede Anwendung in ihrem eigenen Adreßsraum ohne gemeinsame Instanzdaten. Dadurch erscheint jede Instanz als Erstinstanz. Daher wird bei jedem Start einer 32-Bit-Anwendung sowohl die Initialisierung für eine Erstinstanz als auch die für eine beliebige Instanz ausgeführt.

Wenn die aktuelle Instanz eine Erstinstanz ist (angezeigt durch das Datenelement *hPrevInstance*, das auf Null gesetzt ist), wird *InitApplication* aufgerufen. Sie können *InitApplication* in Ihrer abgeleiteten Anwendungsklasse überschreiben; die vorgegebene *InitApplication* hat keine Funktionalität.

Sie könnten zum Beispiel die Initialisierung der Erstinstanz dazu nutzen, im Titel Ihres Hauptfensters anzuzeigen, daß es sich um eine Erstinstanz handelt. Dafür sollten Sie ein Datenelement mit Namen *WindowTitle* in Ihrer abgeleiteten Klasse hinzufügen. Setzen Sie dann im Konstruktor *WindowTitle* auf "Additional Instance". Überschreiben Sie *InitApplication*, um *WindowTitle* auf "First Instance" zu setzen. Wenn Ihre Anwendung die erste Instanz der Anwendung ist, wird *InitApplication* aufgerufen und überschreibt den vom Konstruktor an *WindowTitle* zugewiesenen Wert. Das folgende Beispiel zeigt einen möglichen Code:

```
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <string.h>

class TTestApp : public TApplication {
public:
     TTestApp(): TApplication("Instance Tester") {
                 strcpy(WindowTitle, "Additional Instance");
                 }

protected:
     char WindowTitle[20];

     void InitApplication() {
                 strcpy(WindowTitle, "First Instance");
                 }
```

```
    void InitMainWindow() {
              SetMainWindow = new TFrameWindow(0, WindowTitle);
              }
    };
int OwlMain(int argc,char* argv[])
{
    return TTestApp("Wow!").Run();
}
```

Es sei nochmals erwähnt, daß diese Anwendung nicht funktioniert, wie Sie vielleicht erwarten könnten, wenn sie als 32-Bit-Anwendung erzeugt wurde. Da jede Instanz einer 32-Bit-Anwendung sich selbst als Erstinstanz ansieht, würden mehrere Kopien, die zur gleichen Zeit laufen, alle den Titel "First Instance" haben.

Initialisieren jeder neuen Instanz

Ein Benutzer kann mehrere Instanzen (Kopien) einer Anwendung gleichzeitig laufen lassen. Sie können *TApplication::InitInstance* überschreiben, um jede beliebige Initialisierung durchzuführen, die Sie für jede Instanz benötigen.

InitInstance ruft *InitMainWindow* auf, und danach erzeugt und zeigt es das Hauptfenster, das Sie in *InitMainWindow* aufgebaut haben. Wenn Sie *InitInstance* überschreiben, sollten Sie sich vergewissern, daß Ihre neue Funktion *TApplication::InitInstance* aufruft. Das folgende Beispiel zeigt die Verwendung von *InitInstance*, um eine Tastenkürzeltabelle zu laden.

```
void TTestApp::InitInstance() {
    TApplication::InitInstance();
    HAccTable = LoadAccelerators(MAKEINTRESOURCE(MYACCELS));
    }
```

Initialisieren der Hauptfenster

TApplication::InitMainWindow erzeugt standardmäßig ein Rahmenfenster mit dem gleichen Namen wie das Anwendungsobjekt. Dieses Fenster ist kaum verwendbar, da es keine Benutzereingaben empfangen oder verarbeiten kann. Sie müssen *InitMainWindow* überschreiben, um ein Fensterobjekt zu erzeugen, welches Benutzereingaben verarbeiten kann. Normalerweise erzeugt Ihre *InitMainWindow*-Funktion ein von *TFrameWindow* oder von *TFrameWindow* abgeleitetes Objekt und ruft die Funktion *SetMainWindow* auf. *SetMainWindow* erwartet einen Parameter (ein *TFrameWindow* *) und gibt einen Zeiger auf das alte Hauptfenster zurück (wenn es sich um eine neue Anwendung handelt, bei der noch kein Hauptfenster eingerichtet wurde, ist der Rückgabewert Null).In Kapitel 6 werden Fensterklassen und -objekte eingehender beschrieben.

Das folgende Beispiel zeigt eine einfache Anwendung, die ein *TFrameWindow*-Objekt erzeugt und zum Hauptfenster werden läßt:

```
#include <owl\applicat.h>
#include <owl\framewin.h>

class TMyApplication: public TApplication
{
public:
     TMyApplication(): TApplication() {}
     virtual void InitMainWindow();
};

void TMyApplication::InitMainWindow() {
     SetMainWindow = (new TFrameWindow(0, "My First Main Window"));
     }

int OwlMain(int argc,char* argv[])
{
     return TMyApplication("Wow!").Run();
}
```

Wenn Sie diese Anwendung laufen lassen, zeigt die Titelleiste "My First Main Window" und nicht "My First Application". Der im *TApplication*-Konstruktor übergebene Anwendungsname wird nur dann benutzt, wenn Sie kein Hauptfenster vorgesehen haben. Es sei nochmals erwähnt, daß dieses Beispiel nicht viel bewirkt; es gibt für das Rahmenfenster noch keine Vorkehrungen, um irgendwelche Benutzereingaben zu verarbeiten. Wenn Sie jedoch einmal eine Fensterklasse abgeleitet haben, die auf den Benutzer reagiert, benutzen Sie dieselbe einfache Methode, um das Fenster zu zeigen.

Festlegen des Anzeigemodus für das Hauptfenster

Die Darstellungsweise der Hauptfenster Ihrer Anwendung können Sie ändern, indem Sie das Datenelement *nCmdShow* von *TApplication* entsprechend setzen. Dies korrespondiert zum *WinMain*-Parameter *CmdShow*. Sie können diese Variable setzen, sobald die *Run*-Funktion beginnt bis hin zu einer Zeit vor dem Aufruf von *TApplication::InitInstance*. Das bedeutet konkret, daß Sie *CmdShow* entweder in der *InitApplication*- oder in der *InitMainWindow*-Funktion setzen können.

Nehmen wir beispielsweise an, Sie wollen Ihr Fenster immer in voller Ausdehnung zeigen, wenn der Benutzer die Anwendung laufen läßt. Sie könnten *nCmdShow* in Ihrer *InitMainWindow*-Funktion setzen:

```
#include <owl\applicat.h>
#include <owl\framewin.h>

class TMyApplication : public TApplication {
public :
     TMyApplication(char far *name) : TApplication(name) {}
     void InitMainWindow();
     };
```

```
void TMyApplication::InitMainWindow() {
     SetMainWindow = /new TFrameWindow(0, "Maximum Window"));
     nCmdShow = SW_SHOWMAXIMIZED;
     }

int OwlMain(int argc;char* argv[])
{
     return TMyApplication("Wow!").Run();
}
```

nCmdShow kann auf jeden Wert gesetzt werden, den Sie normalerweise *CmdShow* zuweisen würden (wie zum Beispiel SW_HIDE, SW_SHOWNORMAL, SW_NORMAL usw.

Wechseln des Hauptfensters

Sie können Ihr Hauptfenster im Laufe Ihrer Anwendung wechseln. Verwenden Sie dazu die Funktion *TApplication::SetMainWindow*. *SetMainWindow* erwartet einen Parameter, ein *TFrameWindow* *, und gibt einen Zeiger auf das alte Hauptfenster zurück (wenn es sich um eine neue Anwendung handelt, bei der noch kein Hauptfenster eingerichtet wurde, ist der Rückgabewert Null). Sie können das alte Hauptfenster für den Fall behalten, daß Sie es wiederherstellen wollen. Alternativ können Sie diesen Zeiger auch verwenden, um das alte Hauptfensterobjekt zu löschen.

Botschaftsverarbeitung von Anwendungen

Wenn Ihre Anwendung einmal initialisiert ist, beginnt die *MessageLoop* Ihres Anwendungsobjektes zu laufen. Die *MessageLoop* ist für die Verarbeitung der von Windows hereinkommenden Botschaften zuständig. Es gibt zwei Möglichkeiten, die Botschaftsverarbeitung in einer ObjectWindows-Anwendung auszubauen:

- Gesonderte Botschaftsverarbeitung durch das Überschreiben von Standardfunktionen zur Botschaftsverarbeitung
- Idle-Verarbeitung

Gesonderte Botschaftsverarbeitung

TApplication enthält eine Reihe von Funktionen, die die Basisfunktionalität einer Botschaftsverarbeitung für jede ObjectWindows-Anwendung bereitstellen.

Idle-Verarbeitung

Bei der Idle-Verarbeitung nutzt Ihre Anwendung die Idle-Zeit (Leerlaufzeit), in der keine Botschaften auf Bearbeitung warten (einschließlich der Benutzereingaben). Wenn es keine wartenden Botschaften gibt, ruft *MessageLoop* die Funktion *IdleAction* auf.

Um eine spezielle Idle-Verarbeitung auszuführen, müssen Sie *IdleAction* überschreiben. Denken Sie daran, daß Idle-Verarbeitung nur stattfindet, solange der Benutzer nicht agiert. Daher sollte Idle-Verarbeitung nur von kurzer Dauer sein. Wenn Sie Aktionen planen, die länger als einige Zehntelsekunden benötigen, sollten Sie diese in mehrere Prozesse aufteilen.

Der Parameter von *IdleAction* (*idleCount*) ist ein **long** und gibt die Zahl an, wie oft *IdleAction* zwischen Botschaften aufgerufen wurde. Sie können *idleCount* dazu verwenden, um zwischen Idle-Verarbeitung niedriger und hoher Priorität zu wählen. Wenn *idleCount* einen hohen Wert erreicht, können Sie daraus schließen, daß eine lange Zeitspanne ohne Benutzereingaben vorlag. Damit ist es auch möglich, Idle-Verarbeitung mit niedriger Priorität auszuführen.

Geben Sie bei *IdleAction* den Wert TRUE zurück, damit *IdleAction* zu einem früheren Zeitpunkt wieder aufgerufen wird.

Zusätzlich zur Ausführung Ihrer eigenen Verarbeitung sollten Sie auch immer die *IdleAction*-Funktion der Basisklasse aufrufen. Wenn Sie Anwendungen für Windows NT schreiben, können Sie auch mehrfache Threads für die Hintergrundverarbeitung verwenden.

Schließen von Anwendungen

Benutzer schließen eine Windows-Anwendung üblicherweise, indem Sie *File/Exit* wählen oder [*Alt*] + [*F4*] drücken. Dabei ist es wichtig, daß die Anwendung einen solchen Versuch abfangen kann, damit der Benutzer alle noch offenen Dateien sichern kann. *TApplication* sieht dies vor.

Verhalten beim Schließen ändern

TApplication und alle Fensterklassen besitzen oder erben eine Elementfunktion *CanClose*.Wenn eine Anwendung enden will, fragt sie stets die *CanClose*-Funktion des Hauptfensters und des Dokumentmanagers ab. Wenn für die beiden untergeordnete Objekte vorliegen, ruft sie auch jeweils deren *CanClose*-Funktionen auf. Diese untergeordneten Objekte rufen wiederum die *CanClose*-Funktion jeder der ihnen untergeordneten Objekte auf usw.

Die *CanClose*-Funktion gibt damit jedem Objekt die Möglichkeit, sich auf das Beenden der Anwendung vorzubereiten. Jedes Objekt hat damit zudem auch die Möglichkeit falls notwendig das Beenden abzubrechen. Wenn das Objekt seine Prozedur zum endgültigen Aufräumen abgeschlossen hat, sollte seine *CanClose*-Funktion den Wert TRUE zurückliefern.

Wenn irgendeine der aufgerufenen *CanClose*-Funktionen FALSE zurückgibt, wird die Prozedur zum Beenden der Anwendung abgebrochen.

Die Anwendung schließen

Der *CanClose*-Mechnismus versetzt das Anwendungsobjekt, das Hauptfenster und alle anderen Fenster in die Lage, sich entweder auf das Beenden vorzubereiten oder das Schließen zu verhindern. Zuletzt erkennt das Anwendungsobjekt das Schließen der Anwendung an. Die normale Sequenz für das Schließen sieht ungefähr wie folgt aus:

1. Windows sendet an das Hauptfenster eine WM_CLOSE-Botschaft
2. Die *EvClose*-Elementfunktion des Hauptfensterobjektes ruft die *CanClose*-Elementfunktion des Anwendungsobjektes auf.
3. Die *CanClose*-Elementfunktion des Anwendungsobjektes ruft die *CanClose*-Elementfunktion des Hauptfensterobjektes auf.
4. Das Hauptfensterobjekt und das Dokumentmanagerobjekt rufen *CanClose* für jedes ihrer untergeordneten Fenster auf. Die *CanClose*-Funktion des Hauptfensterobjektes und des Dokumentmanagerobjektes liefern nur dann TRUE zurück, wenn auch *CanClose*-Elementfunktionen aller untergeordneten Fenster TRUE zurückliefern.
5. Wenn die *CanClose*-Funktion sowohl des Hauptfensterobjektes als auch des Dokumentmanagerobjektes TRUE zurückliefert, liefert auch die *CanClose*-Funktion des Anwendungsobjektes TRUE zurück.
6. Wenn die *CanClose*-Funktion des Anwendungsobjektes TRUE zurückliefert, schließt die *EvClose*-Funktion das Hauptfenster und beendet die Anwendung.

Ändern von CanClose

CanClose sollte nur in seltenen Fällen FALSE zurückliefern. Stattdessen sollte *CanClose* alle notwendigen Aktionen durchführen, damit TRUE zurückgegeben werden kann. *CanClose* sollte nur dann FALSE zurückliefern, wenn die notwendigen Befehle für einen ordnungsgemäßen Abschluß nicht durchführbar sind oder wenn der Benutzer das Fortfahren der Anwendung wünscht.

So könnte zum Beispiel die *CanClose*-Elementfunktion eines Editierfensters überprüfen, ob der Editiertext verändert wurde, und dann den Benutzer fragen, ob der Text zum Schließen gespeichert werden soll. Ein Meldungsfenster mit den Schaltern *Ja*, *Nein* und *Abbruch* bietet sich hierfür an. *Abbruch* würde anzeigen, daß der Benutzer die Anwendung noch nicht schließen will. *CanClose* würde dann FALSE zurückliefern.

Öffnen vordefinierter DLLs

TApplication besitzt eine Reihe von Funktionen für den Zugriff auf viele vordefinierte DLLs: die Bibliothek für Dialogelemente im Borland-Stil

(BWCC.DLL für 16-Bit-Anwendungen und BWCC32.DLL für 32-Bit-Anwendungen) und die Bibliothek für 3D-Dialogelemente von Microsoft (CTL3D.DLL). Diese DLLs werden sehr oft benutzt, um vielen Anwendungen standardisiertes Aussehen und Bedienung zu verleihen.

Verwenden der Bibliothek für Dialogelemente im Borland-Stil (BWCC)

Über die Funktion *TApplication::EnableBWCC* können Sie die Bibliothek für Dialogelemente im Borland-Stil öffnen und schließen. *EnableBWCC* erwartet ein BOOL als Parameter und gibt **void** zurück. Wenn Sie TRUE an *EnableBWCC* übergeben, öffnet die Funktion die DLL, sofern sie nicht schon geöffnet ist. Wenn Sie FALSE an *EnableBWCC* übergeben, schließt die Funktion die DLL, sofern sie nicht schon geschlossen ist. Über die Funktion *TApplication::BWCCEnabled* können Sie in Erfahrung bringen, ob die DLL der Bibliothek für Dialogelemente im Borland-Stil (BWCC) geöffnet oder geschlossen ist. *BWCCEnabled* hat keine Parameter. Die Funktion gibt TRUE zurück, wenn die DLL geöffnet ist, und FALSE, wenn sie geschlossen ist.

Wenn eine DLL einmal geöffnet ist, können Sie die gesamte reguläre Funktionalität der Bibliothek für Dialogelemente im Borland-Stil nutzen. Beachten Sie, daß *EnableBWCC* abhängig davon, ob es sich um eine 16- oder 32-Bit-Anwendung handelt, die korrekte Bibliothek öffnet.

Verwenden der Bibliothek für 3D-Dialogelemente von Microsoft

Über die Funktion *TApplication::EnableCtl3d* können Sie die Bibliothek für 3D-Dialogelemente von Microsoft öffnen. *EnableCtl3d* erwartet ein BOOL als Parameter und gibt **void** zurück. Wenn Sie TRUE an *EnableCtl3d* übergeben, öffnet die Funktion die DLL, sofern sie nicht schon geöffnet ist. Wenn Sie FALSE an *EnableCtl3d* übergeben, schließt die Funktion die DLL, sofern sie nicht schon geschlossen ist. Über die Funktion *TApplication::Ctl3dEnabled* können Sie in Erfahrung bringen, ob die DLL-Bibliothek für 3D-Dialogelemente von Microsoft geöffnet oder geschlossen ist. *Ctl3dEnabled* hat keine Parameter. Die Funktion gibt TRUE zurück, wenn die DLL geöffnet ist, und FALSE, wenn sie geschlossen ist. Um die Funktion *EnableCtl3dAutosubclass* zu benutzen, müssen Sie zuerst die DLL-Bibliothek für 3D-Dialogelemente von Microsoft über *EnableCtl3d* öffnen. *EnableCtl3dAutosubclass* erwartet ein BOOL als Parameter und gibt **void** zurück. Wenn Sie TRUE an *EnableCtl3dAutosubclass* übergeben, wird die automatische Subklasseneinordnung angeschaltet.Ist dies angeschaltet, wird jeder von Ihnen erzeugte nicht-ObjectWindows-Dialog einen 3D-Effekt zeigen. Sie können die automatische Subklasseneinordnung direkt nach der Erzeugung des Dialogfensters abschalten. Es ist nicht notwendig, sie angeschaltet zu lassen, während das Dialogfenster angezeigt wird.

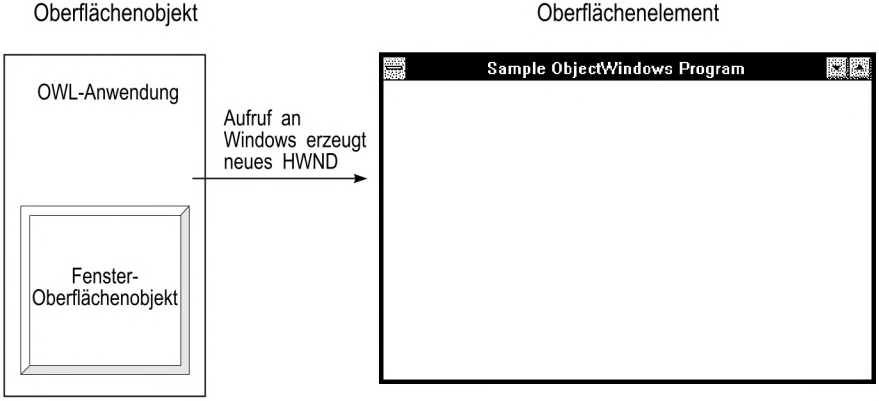
K A P I T E L 4

Oberflächenobjekte

Instanzen von Klassen, die Fenster, Dialogfenster oder Dialogelemente repräsentieren, werden *Oberflächenobjekte* genannt. Dieses Kapitel geht näher auf die allgemeinen Anforderungen und das Verhalten von Oberflächenobjekten sowie auf deren Verhältnis zu den *Oberflächenelementen* – das sind die aktuell auf dem Bildschirm erscheinenden Fenster, Dialogfenster oder Dialogelemente – ein.

Das folgende Diagramm verdeutlicht den Unterschied zwischen Oberflächenobjekten und Oberflächenelementen:

Abbildung 3.1: Oberflächenobjekte und Oberflächenelemente



Beachten Sie, daß das Oberflächenobjekt innerhalb des Anwendungsobjektes plaziert ist. Das Oberflächenobjekt ist eigentlich eine ObjectWindows-Klasse, die im Heap oder Stack der Anwendung erzeugt und gespeichert wird. Dies hängt im einzelnen davon ab, wie Speicher für das Objekt reserviert wird. Das Oberflächenelement ist dagegen ein Bestandteil von Windows. Es ist das eigentliche Fenster, das auf dem Bildschirm dargestellt wird.

Die Informationen dieses Kapitels erstrecken sich auf alle Oberflächenobjekte. Es erläutert ebenso die Verhältnisse zwischen verschiedenen Oberflächenobjekten einer Anwendung sowie den Mechanismus, mit dem Oberflächenobjekte für gewöhnlich auf Windows-Botschaften reagieren.

Warum Oberflächenobjekte?

Eine der größten Schwierigkeiten bei der Windows-Programmierung besteht darin, daß die Kontrolle der Oberflächenelemente inkonsistent und verwirrend sein kann. Manchmal schicken Sie eine Botschaft an ein Fenster, manchmal rufen Sie eine Windows-API-Funktion auf. Die Konventionen für ähnliche Typen von Operationen unterscheiden sich oft, wenn diese Operationen mit unterschiedlichen Typen von Elementen ausgeführt werden.

ObjectWindows erleichtert viele dieser Schwierigkeiten, indem Objekte zur Verfügung gestellt werden, die die Oberflächenelemente einkapseln. Dies erspart Ihnen den direkten Umgang mit Windows und führt zu einer einheitlicheren Schnittstelle für die Kontrolle von Oberflächenelementen.

Was bewirken Oberflächenobjekte?

Ein Oberflächenobjekt besitzt Elementfunktionen für die Erzeugung, Initialisierung, Verwaltung und das Entfernen des mit ihm verbundenen Oberflächenelementes. Die Elementfunktionen übernehmen für Sie viele Details der Windows-Programmierung.

Oberflächenobjekte kapseln zudem die für die Kommunikation mit dem Oberflächenelement benötigten Daten wie Handles und Zeiger zu unter- und übergeordneten Fenstern ein.

Das Verhältnis zwischen Oberflächenobjekten und Oberflächenelementen ist mit dem zwischen einer Datei auf einer Diskette oder Platte und einem C++ Stream-Objekt vergleichbar. Das Stream-Objekt repräsentiert nur die aktuelle Datei auf der Diskette. Sie manipulieren diese Datei, indem Sie das Stream-Objekt manipulieren. Bei ObjectWindows repräsentieren Oberflächenobjekte gerade die Oberflächenelemente, die von Windows aktuell verwaltet werden. Sie arbeiten mit dem Objekt und Windows sorgt für die Verwaltung des Windows-Elementes.

Das generische Oberflächenobjekt: TWindow

Oberflächenobjekte von ObjectWindows werden alle von *TWindow* abgeleitet, die das für alle Fenster-, Dialogfeld- und Dialogelementobjekte ge-

meinsame Verhalten vorgibt. *TFrameWindow*, *TDialog*, und *TControl* sind von *TWindow* abgeleitet und differenzieren das generische Verhalten von *TWindow*, je nach Bedarf.

TWindow ist eine für alle Oberflächenobjekte gemeinsame Basisklasse und gibt damit einheitliche Vorgehensweisen für folgende Aktionen vor:

- Verwaltung der Beziehung zwischen Oberflächenobjekten und Oberflächenelementen einschließlich dem Erzeugen und Entfernen von Objekten und Elementen
- Behandlung von unter- und übergeordneten Verhältnissen zwischen Oberflächenobjekten
- Registrierung von neuen Windows-Fensterklassen

Erzeugen von Oberflächenobjekten

Die Einrichtung eines Oberflächenobjektes mit seinem verbundenen Oberflächenelement erfordert zwei Schritte:

1. Rufen Sie einen der Konstruktoren für Oberflächenobjekte, die das Oberflächenobjekt konstruieren und seine Attribute festlegen, auf.
2. Erzeugen Sie das Oberflächenelement, indem Sie Windows auffordern, das Oberflächenobjekt zusammen mit dem neuen Oberflächenelement zu erzeugen:

   Bei den meisten Oberflächenelementen rufen Sie zur Erzeugung die Elementfunktion *Create* des Oberflächenobjektes auf. *Create* ruft indirekt auch *SetupWindow* auf. Diese Funktion initialisiert ein Oberflächenobjekt durch die Erzeugung eines Oberflächenelementes ähnlich der Erzeugung eines untergeordneten Fensters.

   Bei einem modalen Dialogfenster rufen Sie zur Erzeugung des Oberflächenelementes die Elementfunktion *Execute* des Oberflächenobjektes auf.

Die Verbindung zwischen Oberflächenobjekt und Oberflächenelement wird über das Datenelement *Window* des Oberflächenobjektes verwaltet. Es enthält ein Handle auf ein Fenster.

Wann ist ein Fenster-Handle gültig?

Normalerweise empfängt ein neu erzeugtes Oberflächenelement unter Windows eine WM_CREATE-Botschaft von Windows und antwortet darauf, indem es sich selbst initialisiert. ObjectWindows-Oberflächenobjekte fangen die WM_CREATE-Botschaft ab und rufen stattdessen *SetupWin-*

dow auf. *SetupWindow* ist die Stelle, wo Sie Ihre eigene Initialisierung vornehmen können.

Wenn ein Teil der Initialisierung für das Oberflächenobjekt das Fenster-Handle des Oberflächenelementes benötigt, dann müssen Sie diese Initialisierung *nach* dem Aufruf der Funktion *SetupWindow* aus der Basisklasse durchführen. Vor dem Aufruf der Basisklassenfunktion *SetupWindow* sind das Fenster und seine untergeordneten Fenster noch nicht erzeugt worden; *HWindow* ist nicht gültig und sollte auch nicht benutzt werden. Sie können die Gültigkeit von *HWindow* auf einfache Weise überprüfen: wenn es nicht initialisiert wurde, hat es den Wert NULL.

Wenn es auch ungewöhnlich erscheinen mag, daß Sie nicht die vollständige Initialisierung im Konstruktor des Oberflächenobjektes durchführen können, so gibt es dafür doch gute Gründe: Wenn ein Oberflächenelement einmal erzeugt ist, können viele seiner Charakteristika nicht mehr verändert werden. Deshalb ist eine zweiteilige Initialisierung notwendig: vor und nach der Erzeugung des Oberflächenelementes.

Der Konstruktor des Oberflächenobjektes ist die Stelle für Initialisierungen, die vor der Erzeugung des Elementes stattfinden sollen, und *SetupWindow* ist die Stelle für Initialisierungen nach der Erzeugung. Sie können sich *SetupWindow* auch als den zweiten Teil des Konstruktors vorstellen.

Sichtbarmachen von Oberflächenelementen

Wenn Sie ein Objekt und sein entsprechendes Element erzeugen, bedeutet das nicht, daß Sie schon etwas auf dem Bildschirm sehen. Wenn Windows ein Oberflächenelement erzeugt, wird überprüft, ob die Elementeigenschaften WS_VISIBLE einschließen. Trifft dies zu, wird das Oberflächenelement von Windows auf dem Bildschirm angezeigt. Trifft dies nicht zu, wird das Element erzeugt, aber nicht auf dem Bildschirm dargestellt.

Der Konstruktor von *TWindow* setzt WS_VISIBLE, wodurch die meisten Oberflächenobjekte standardmäßig sichtbar sind. Wenn Ihr Objekt allerdings eine Ressource lädt, hängt die Ressource-Eigenschaft von der Definition in der entsprechenden Ressourcendatei ab. Ist WS_VISIBLE bei der Ressource-Eigenschaft angeschaltet, dann ist auch WS_VISIBLE für das Objekt angeschaltet. Ist WS_VISIBLE bei der Ressource-Eigenschaft *nicht* angeschaltet, dann ist auch WS_VISIBLE für die Objekteigenschaft *abgeschaltet*. Sie können WS_VISIBLE und andere Fenstereigenschaften im Datenelement *Attr.Style* im Oberflächenobjekt festlegen.

Wenn Sie zum Beispiel mit Hilfe von *TDialog* eine Dialog-Ressource laden, die WS_VISIBLE nicht angeschaltet hat, müssen Sie WS_VISIBLE explizit anschalten, bevor Sie den Dialog über *Create* anzeigen können.

Sie können die Sichtbarkeit eines Oberflächenobjektes überprüfen, indem Sie *IsWindowVisible* aufrufen. *IsWindowVisible* liefert TRUE, wenn das Ob-

jekt sichtbar ist.Zu jeder Zeit nach der Erzeugung des Oberflächenelementes können Sie es anzeigen oder verstecken, indem Sie die Elementfunktion *Show* mit dem Wert TRUE bzw. FALSE aufrufen.

Objekteigenschaften

Das Oberflächenobjekt besitzt, zusätzlich zu den Attributen seines Oberflächenelementes, bestimmte Attribute, die es als ObjectWindows-Objekt benötigt. Sie können diese Eigenschaften und Charakteristika mit Hilfe der folgenden Funktionen abfragen und ändern:

- *SetFlag* setzt das angegebene Flag für das Objekt.
- ClearFlag löscht das angegebene Flag für das Objekt.
- IsFlagSet gibt TRUE zurück, wenn das angegebene Flag gesetzt ist, anderenfalls FALSE.

Sie können folgende Flags für diese Funktionen verwenden:

- *wfAlias* gibt an, ob das Objekt ein Alias ist.
- wfAutoCreate gibt an, ob die automatische Erzeugung für dieses Objekt eingeschaltet ist.
- wfFromResource gibt an, ob das Oberflächenelement von einer Ressource geladen wurde.
- wfShrinkToClient gibt an, ob das Rahmenfenster schrumpfen sollte, damit es zu der Größe des Client-Fensters paßt.
- wfMainWindow gibt an, ob das Fenster das Hauptfenster ist.
- wfPredefinedClass gibt an, ob das Fenster eine vordefinierte Windows-Klasse ist.
- wfTransfer gibt an, ob das Fenster den Datentransfermechanismus benutzen kann.

Fenstereigenschaften

TWindow enthält auch eine Reihe von Funktionen, mit deren Hilfe Sie Ressourcen und Eigenschaften des Oberflächenelementes ändern können. Da *TWindow* für viele verschiedene Objekte eine generische Funktionalität bereitstellt, bietet sie keine sehr spezifischen Funktionen für die Manipulation von Ressourcen und Eigenschaften. Objekte, die auf hoher Stufe stehen, bieten wesentlich mehr spezifische Funktionalität. Diese spezifische Funktionalität baut jedoch ein Objekt weiter auf und ist als Zusatz zu der von *TWindow* bereitgestellten Funktionalität anzusehen.

- *SetCaption* setzt den Fenstertitel auf den String, den Sie als Parameter übergeben.
- GetWindowTextTitle gibt einen String zurück, der den aktuellen Fenstertitel enthält.
- SetCursor setzt den Cursor der Instanz, die durch *TModule* als ersten Parameter identifiziert wird, auf den Cursor, der im zweiten Parameter als Ressource übergeben wurde.

- Sie können die Tastenkürzeltabelle für ein Fenster festlegen, indem Sie die Ressource-ID (die ein String oder eine Ganzzahl sein kann) an *Attr.AccelTable* zuweisen. Nehmen wir zum Beispiel an, Sie haben eine Tastenkürzeltabelle als Ressource mit Namen MY_ACCELS. Dann könnten Sie die Ressource an *Attr.AccelTable* wie folgt zuweisen:

```
TMyWnd::TMyWnd(const char* title)
     : TFrameWindow(0, title),
       TWindow(0, title) {
     Attr.AccelTable = OWLHELPAPACCEL; // Tastenkürzeltabelle kann
// zugewiesen werden
     }
```

Weiterführende Informationen über diese Funktionen finden Sie im *ObjectWindows Referenzhandbuch*.

Entfernen von Oberflächenobjekten

Das Entfernen eines Oberflächenobjektes besteht aus zwei Schritten:

- Entfernen des Oberflächenelementes
- Löschen des Oberflächenobjektes

Die beste Vorgehensweise dazu besteht in dem Aufruf der Elementfunktion *ShutDownWindow*. *ShutDownWindow* entfernt das Element und löscht das Objekt in einem Schritt. Sie rufen die *ShutDownWindow*-Funktion für Ihr Oberflächenobjekt wie folgt auf:

```
TWindow *window = new TWindow(0, "My Window");

// weitere Aktionen hier.

window->ShutDownWindow();
```

Sie können Ihr Objekt auch in zwei separaten Schritten entfernen und löschen. Dies ist besonders dann nützlich, wenn Sie ein Oberflächenelement entfernen und vom Bildschirm löschen, das Oberflächenobjekt aber weiter behalten wollen, um es später wieder darzustellen.

Entfernen des Oberflächenelementes

Wenn Sie vorhaben, das Oberflächenelement erneut zu erzeugen und anzuzeigen, können Sie es entfernen, ohne das Oberflächenobjekt zu löschen. Beachten Sie, daß diese Methoden nur geeignet funktionieren sofern das *TWindow*-Objekt über **new** erzeugt wurde.

Für die Entfernung eines Oberflächenelementes ist die Elementfunktion *Destroy* des Oberflächenobjektes zuständig. *Destroy* entfernt die Oberflächenelemente über den Aufruf der API-Funktion *DestroyWindow* (es sei denn, das Objekt ist ein Ersatz. In diesem Fall setzt *Destroy* den Wert von *HWindow* einfach auf Null).. Wenn das Oberflächenelement entfernt ist,

ist das Datenelement *HWindow* des Oberflächenobjektes auf Null gesetzt. Deswegen können Sie auch über das Datenelement *HWindow* eines Oberflächenobjektes feststellen, ob es noch mit einem gültigen Oberflächenelement verbunden ist.

Wenn ein Benutzer ein Fenster auf dem Bildschirm schließt, werden folgende Schritte ausgelöst:

- Windows benachrichtigt das Fenster
- Das Fenster durchläuft den *CanClose*-Mechanismus, um festzustellen, daß das Fenster geschlossen werden sollte.
- Wenn *CanClose* das Schließen des Fensters bestätigt, wird das Oberflächenelement entfernt und das Oberflächenobjekt gelöscht.

Löschen des Oberflächenobjektes

Wenn Sie selber ein Oberflächenelement entfernen und somit das Oberflächenobjekt für spätere Anzeigen behalten können, müssen Sie sicherstellen, daß Sie das Oberflächenobjekt löschen, wenn Sie es nicht mehr benötigen. Da ein Oberflächenobjekt nichts anderes als ein reguläres C++ Objekt ist, können Sie es mit der **delete**-Anweisung löschen.

Untergeordnete und übergeordnete Oberflächenelemente

In einer Windows-Anwendung arbeiten Oberflächenelemente in einer über- und untergeordneten Verbindung zusammen. Ein übergeordnetes Fenster kontrolliert seine untergeordneten Fenster und Windows hält diese Verbindungen aufrecht. ObjectWindows verwaltet eine parallele Menge von Verbindungen zwischen korrespondierenden Oberflächenobjekten.

Ein untergeordnetes Fenster ist ein Oberflächenelement, das durch ein anderes Oberflächenelement gesteuert wird. So werden zum Beispiel Auswahllisten durch das Fenster oder Dialogfenster gesteuert, in dem sie erscheinen. Sie werden nur dann angezeigt, wenn auch ihre übergeordneten Fenster angezeigt werden. Dialogfenster wiederum sind untergeordnete Fenster und werden von den Fenstern gesteuert, von denen sie erzeugt wurden.

Wenn Sie ein übergeordnetes Fenster bewegen oder schließen, werden die untergeordneten Fenster automatisch mitbewegt oder geschlossen.

Das Hauptfenster ist das höchste Objekt in einer Anwendung, dem alle anderen Fenster untergeordnet sind (es gibt allerdings eine Reihe von Ausnahmen: Es kann Fenster und Dialogfenster geben, die keinem Objekt untergeordnet sind; ferner sind alle Hauptfenster dem Windows-Desktop untergeordnet).

Listen untergeordneter Fenster

Wenn Sie ein untergeordnetes Fensterobjekt konstruieren, geben Sie das übergeordnete Fenster als ein Parameter für den Konstruktor an. Ein untergeordnetes Fenster speichert sein übergeordnetes Fenster in dem Datenelement *Parent*. Ein übergeordnetes Fenster speichert seine untergeordneten Fenster in einem privaten Datenelement mit Namen *ChildList*. Jedes übergeordnete Fenster verwaltet seine Liste mit untergeordneten Fenstern automatisch.Sie können auf die untergeordneten Fenster eines Objektes über die fenster-iterierenden Elementfunktionen *FirstThat* und *ForEach* zugreifen.

Konstruieren von untergeordneten Fenstern

Wie bei allen Oberflächenobjekten werden auch untergeordnete Fensterobjekte in zwei Schritten erzeugt: Konstruieren des Oberflächenobjektes und Erzeugen des Oberflächenelementes. Wenn Sie untergeordnete Fensterobjekte im Konstruktor des übergeordneten Fensters konstruieren, werden ihre Oberflächenelemente automatisch erzeugt; vorausgesetzt, daß im übergeordneten Fenster die automatische Erzeugung für untergeordnete Fenster eingeschaltet ist. Standardmäßig ist die automatische Erzeugung für alle ObjectWindows-Objekte, die auf *TWindow* basieren, eingeschaltet. Eine Ausnahme bildet allerdings *TDialog*.

Ein Konstruktor für ein von *TWindow* abgeleitetes Fensterobjekt, das drei untergeordnete Schalterfenster enthält, könnte zum Beispiel wie folgt aussehen:

```
TTestWindow::TTestWindow(TWindow *parent, const char far *title)
   : TFrameWindow(parent, title),
     TWindow(parent, title) {
   button1 = new TButton(this, ID_BUTTON1, "Show",
                         190, 270, 65, 20, FALSE);
   button2 = new TButton(this, ID_BUTTON2, "Hide",
                         275, 270, 65, 20, FALSE);
   button3 = new TButton(this, ID_BUTTON3, "Transfer",
                         360, 270, 65, 20, FALSE);
   }
```

Beachten Sie die Verwendung des **this**-Zeigers, um die untergeordneten Fenster mit dem übergeordneten zu verbinden. Die Konstruktoren von Oberflächenobjekten fügen sich selbst automatisch in die Liste untergeordneter Fenster bei ihrem übergeordneten Fenster ein. Wenn eine Instanz von *TTestWindow* erzeugt wird, werden die drei Schalter automatisch im Fenster angezeigt.

Erzeugen von untergeordneten Oberflächenelementen

Wenn sie untergeordnete Fensterobjekte nicht im Konstruktor ihres übergeordneten Fensterobjektes konstruieren, werden sie nicht automatisch erzeugt und mit ihrem übergeordneten Objekt angezeigt. Sie können sie dann selbst über *Create* bzw. im Falle von modalen Dialogfenstern über *Execute* erzeugen. In diesem Zusammenhang bedeutet Erzeugen die Instantiierung eines Oberflächenelementes.

Nehmen wir beispielsweise an, daß mit der Erzeugung des Hauptfensters zwei Schalter angezeigt werden, einer mit der Beschriftung *Show* und der andere mit *Hide*. Wenn der Benutzer den *Show*-Schalter drückt, wollen Sie einen dritten Schalter mit der Beschriftung *Transfer* anzeigen. Wenn der Benutzer den *Hide*-Schalter drückt, wollen Sie den *Transfer*-Schalter entfernen:

```
class TTestWindow : public TFrameWindow {
   TButton *button1, *button2, *button3;
public:
   TTestWindow(TWindow *parent, const char far *title);
   void EvButton1() {
      if(!button3->HWindow) {
         button3->Create();
         }
      }
   void EvButton2() {
      if(button3->HWindow)
         button3->Destroy();
      }
   void EvButton3() {
      MessageBeep(-1);
      }
   DECLARE_RESPONSE_TABLE(TTestWindow);
   };
DEFINE_RESPONSE_TABLE1(TTestWindow, TFrameWindow)
   EV_COMMAND(ID_BUTTON1, EvButton1),
   EV_COMMAND(ID_BUTTON2, EvButton2),
   EV_COMMAND(ID_BUTTON3, EvButton3),
END_RESPONSE_TABLE;
TTestWindow::TTestWindow(TWindow *parent, const char far *title)
   : TFrameWindow(parent, title),
     TWindow(parent, title) {
   button1 = new TButton(this, ID_BUTTON1, "Show",
                         10, 10, 75, 25, FALSE);
   button2 = new TButton(this, ID_BUTTON2, "Hide",
                         95, 10, 75, 25, FALSE);
   button3 = new TButton(this, ID_BUTTON3, "Transfer",
                         180, 10, 75, 25, FALSE);
   button3->DisableAutoCreate();
   }
```

Der Aufruf von *DisableAutoCreate* im Konstruktor verhindert, daß der *Transfer*-Schalter beim Erzeugen von *TTestWindow* angezeigt wird. Die Tests der bedingten Anweisungen in den Funktionen *EvButton1* und *EvButton2* nutzen den Test auf Gültigkeit des Datenelementes *HWindow* vom Oberflächenobjekt *button3* aus. Wenn der *Transfer*-Schalter bereits angezeigt wird, versucht *EvButton1* keine erneute Anzeige. *EvButton2* versucht dagegen nicht, den *Transfer*-Schalter zu entfernen, wenn er nicht angezeigt wird.

Entfernen von Fenstern

Mit dem Entfernen eines übergeordneten Fensters werden auch alle seine untergeordneten Fenster entfernt. Sie müssen nicht explizit alle untergeordneten Fenster entfernen oder deren Oberflächenobjekte löschen. Dasselbe trifft auch auf den *CanClose*-Mechanismus zu. *CanClose* für ein übergeordnetes Fenster ruft auch *CanClose* für alle seine untergeordneten Fenster auf. *CanClose* für ein übergeordnetes Fenster gibt nur dann TRUE zurück, wenn auch die *CanClose*-Funktionen für alle untergeordneten Fenster TRUE zurückgeben.

Wenn Sie das Oberflächenelement eines Objektes entfernen, wird die automatische Erzeugung für alle untergeordneten Objekte eingeschaltet, *ungeachtet* dessen, ob automatisches Erzeugen vorher angeschaltet war oder nicht. Dadurch werden bei der Erzeugung eines übergeordneten Objektes alle untergeordneten Objekte in den Zustand wieder zurückversetzt, in dem sie sich vor der Entfernung ihres übergeordneten Objektes befanden.

Um dies zu verhindern, müssen Sie das automatische Erzeugen für alle untergeordneten Objekte, für die Sie es wünschen, explizit abschalten.

Automatisches Erzeugen

Wenn das automatische Erzeugen für ein untergeordnetes Oberflächenobjekt angeschaltet ist, bevor dessen übergeordnetes Objekt erzeugt wurde, wird das untergeordnete Objekt gleichzeitig mit dem übergeordneten erzeugt. Dies trifft auf alle untergeordneten Objekte eines übergeordneten Objektes zu.

Um ein untergeordnetes Fenster vom automatischen Mechanismus zum Erzeugen und Anzeigen explizit auszuschließen, können Sie die Elementfunktion *DisableAutoCreate* im Konstruktor des untergeordneten Objektes aufrufen. Um ein untergeordnetes Fenster zum automatischen Mechanismus zum Erzeugen und Anzeigen explizit hinzuzufügen (zum Beispiel ein Dialogfenster, das normalerweise ausgeschlossen ist), können Sie die Elementfunktion *EnableAutoCreate* im Konstruktor des untergeordneten Objektes aufrufen.

Standardmäßig ist das automatische Erzeugen für alle ObjectWindows-Klassen eingeschaltet. Eine Ausnahme stellen hierbei die Dialogfenster dar.

Manipulieren von untergeordneten Fenstern

TWindow stellt zwei Iteratorfunktionen bereit: *ForEach* und *FirstThat*. Sie ermöglichen die Ausführung von Operationen entweder auf allen untergeordneten Fenstern in der Liste des übergeordneten Fensters auf einmal oder nur auf einem einzelnen untergeordneten Fenster. *TWindow* enthält zudem eine Zahl anderer Funktionen, die es Ihnen ermöglichen, die Zahl der untergeordneten Fenster in der Liste zu bestimmen, sich einzeln durch die Liste zu bewegen oder sich an den Anfang bzw. an das Ende der Liste zu bewegen.

Operieren auf allen untergeordneten Fenstern: ForEach

Es kann vorkommen, daß Sie auf jedem der zu einem Fenster gehörenden untergeordneten Fenster eine Operation ausführen wollen. Die Iteratorfunktion *ForEach* erwartet einen Zeiger auf eine Funktion. Diese Funktion kann entweder eine Elementfunktion oder eine alleinstehende Funktion sein. Die Funktion sollte als Argument ein *TWindow* * und ein **void** * aufnehmen. *ForEach* ruft die Funktion für jedes untergeordnete Fenster einmal auf. Das untergeordnete Fenster wird als *TWindow* * übergeben. **void** * enthält standardmäßig Null. Sie können **void** * dazu benutzen, um beliebige Argumente zu übergeben, die Sie für Ihre Funktion noch benötigen.

Wenn *ForEach* Ihre Funktion aufgerufen hat, sollten Sie im Umgang mit dem untergeordneten Objekt vorsichtig sein. Wenn das Objekt auch als ein *TWindow* * übergeben wurde, ist es für gewöhnlich im aktuellen Fall ein Nachkomme von *TWindow*. Um die korrekte Behandlung des untergeordneten Objektes sicherzustellen, sollten sie mit Hilfe des Makros DYNAMIC_CAST *TWindow* * in *TClass* * umwandeln. Dabei richtet sich *TClass* nach dem Typ des untergeordneten Objektes.

Nehmen wir zum Beispiel an, Sie wollten alle untergeordneten Fenster für Markierungsfelder in einem übergeordneten Fenster überprüfen:

```
void CheckTheBox(TWindow* win, void*) {
     TCheckbox *cb = DYNAMIC_CAST(win, TCheckbox);
     if(cb)
               cb->Check();
     }

void TMDIFileWindow::CheckAllBoxes() {
     ForEach(CheckTheBox);
     }
```

Wenn eine Klasse, auf die hinunter Sie umwandeln, von ihrer Basisklasse virtuell abgeleitet ist (in diesem Fall von *TWindow* nach *TCheckbox*), *müssen* Sie das Makro DYNAMIC_CAST für die Zuweisung verwenden. In dem vorliegenden Fall ist *TCheckbox* von *TWindow* nicht virtuell abgeleitet. Deshalb ist das Makro DYNAMIC_CAST in einem gewissen Sinn überflüssig.

DYNAMIC_CAST gibt Null zurück, wenn die Umwandlung nicht ausgeführt werden konnte. Dies ist hier von Nutzen, da nicht alle unsere unter-

geordneten Objekte notwendigerweise vom Typ *TCheckbox* sind. Wenn ein untergeordnetes Objekt vom Typ *TControlbar* vorgefunden würde, hätte *cb* den Wert Null. Damit ist sichergestellt, daß wir nicht die Überprüfung einer Dialogelementleiste versuchen.

Auffinden eines speziellen untergeordneten Fensters

Es kann ebenso vorkommen, daß Sie eine Funktion nur für ein bestimmtes untergeordnetes Fenster ausführen wollen. Wenn Sie zum Beispiel in einem übergeordneten Fenster mit mehreren Markierungsfeldern das erste markierte finden wollen, können Sie *TWindow::FirstThat* benutzen:

```
BOOL IsThisBoxChecked(TWindow* win, void*) {
   return cb ?
      (cb->GetCheck == BF_CHECKED) :
      FALSE;
   }

TCheckBox* TMDIFileWindow::GetFirstChecked() {
   return FirstThat(IsThisBoxChecked);
   }
```

Arbeiten mit der Liste untergeordneter Fenster

Zusätzlich zu den Iteratorfunktionen *ForEach* und *FirstThat* bietet *TWindow* eine Anzahl von Funktionen, die Ihnen den Zugriff und die Manipulation eines einzelnen untergeordneten Fensters ermöglichen:

- *NumChildren* gibt ein **unsigned** zurück. Dieser Wert gibt die absolute Anzahl von untergeordneten Fenster in der Liste an.
- GetFirstChild gibt ein *TWindow* * zurück, der auf den ersten Eintrag in der Liste untergeordneter Fenster zeigt.
- GetLastChild gibt ein *TWindow* * zurück, der auf den letzten Eintrag in der Liste untergeordneter Fenster zeigt.
- Next gibt ein *TWindow* * zurück, der auf den nächsten Eintrag in der Liste untergeordneter Fenster zeigt.
- Previous gibt ein *TWindow* * zurück, der auf den vorherigen Eintrag in der Liste untergeordneter Fenster zeigt.

Registrieren von Fensterklassen

Wenn Sie ein Oberflächenelement über die Funktionen *Create* oder *Execute* von einem Oberflächobjekt aus erzeugen, überprüft das Objekt, ob ein anderes Objekt desselben Typs mit Windows registriert wurde. Ist dies der Fall, wird das Element erzeugt, basierend auf der existierenden Windows-Registrierklasse. Ist dies nicht der Fall, registriert das Objekt sich automatisch selbst und wird dann auf der Basis derjenigen Klasse erzeugt, die gerade registriert wurde.

Dies nimmt dem Programmierer die Last ab, sicherzustellen, daß alle Fensterklassen vor ihrer Verwendung registriert wurden.

K A P I T E L 5

Ereignisbehandlung

Dieses Kapitel beschreibt, wie die Antworttabellen von ObjectWindows 2.5 angewendet werden. Antworttabellen stellen die Methode dar, mit der alle Ereignisse behandelt werden, die in einer ObjectWindows 2.5 Anwendung auftreten können. Für eine Verwendung der Antworttabellen von ObjectWindows sind vier Hauptschritte erforderlich:

- Deklarieren der Antworttabelle
- Definieren der Antworttabelle
- Definieren der Antworttabelleneinträge
- Deklarieren und Definieren der Antwortelementfunktionen

Um eines der in diesem Kapitel beschriebenen Makros zu verwenden, müssen Sie die Header-Datei OWL\EVENTHAN.H einbinden. Diese Datei ist bereits durch OWL\MODULE.H (die durch OWL\APPLICAT.H eingebunden ist) und OWL\WINDOW.H eingebunden. Deshalb besteht üblicherweise kein Bedarf, diese Datei explizit einzubinden.

ObjectWindows 2.5 Antworttabellen stellen eine erhebliche Verbesserung gegenüber anderen Methoden zur Behandlung der Ereignisse und Botschaften von Windows dar, einschließlich der **switch**-Anweisungen (wie solche in Standard C-Programmen für Windows) und der Schemata, die in anderen Typen von Anwendungsentwicklungen verwendet werden. Im Gegensatz zu jeder anderen Methode zur Ereignisbehandlung bieten ObjectWindows 2.5 Antworttabellen folgendes:

- Automatisches Botschaftsextrahieren für vordefinierte Kommandobotschaften, wodurch ein manuelles Extrahieren der Werte WPARAM und LPARAM nicht notwendig ist.
- Fehler- und Typprüfung zum Zeitpunkt der Compilierung, die den Rückgabetyp und die Parametertypen der Funktion zur Ereignisbehandlung testet.
- Fähigkeit, über eine Funktion mehrere Botschaften zu behandeln.

- Unterstützung der mehrfachen Vererbung, wodurch jede abgeleitete Klasse auf Basisklassen bzw. auf deren Antworttabellen aufbauen können.
- Kein Aufbau auf produktspezifischen Compiler-Erweiterungen, wodurch eine größere Portabilität zwischen verschiedenen Umgebungen erreicht wird.
- Einfache Behandlung von Kommandobotschaften, registrierten Botschaften, Benachrichtigungen, die auf der ID untergeordneter Fenster basieren, und von selbstdefinierten Botschaften. Diese Behandlung erfolgt mit Hilfe der vordefinierten Antworttabellenmakros.

Deklarieren von Antworttabellen

Da die Antworttabelle ein Element einer Klasse von ObjectWindows ist, müssen Sie die Antworttabelle deklarieren, wenn Sie die Klasse definieren. ObjectWindows stellt das Makro DECLARE_RESPONSE_TABLE bereit, um die von Antworttabellen verwendete aktuelle Template-Syntax zu verstecken.

Das Makro DECLARE_RESPONSE_TABLE erwartet ein einziges Argument, den Namen der Klasse, für die die Antworttabelle deklariert wird. Fügen Sie das Makro ans Ende Ihrer Klassendefinition hinzu. Zum Beispiel würde *TMyFrame*, die von *TFrameWindow* abgeleitet ist, wie folgt definiert werden:

```
class TMyFrame : public TFrameWindow
{
  :
  DECLARE_RESPONSE_TABLE(TMyFrame);
  };
```

Definieren von Antworttabellen

Wenn Sie eine Antworttabelle einmal deklariert haben, müssen Sie sie auch definieren. Antworttabellendefinitionen müssen außerhalb der Klassendefinition stehen.

ObjectWindows stellt das Makro DEFINE_RESPONSE_TABLE*X* zur Unterstützung der Definition von Antworttabellen bereit.

Der Wert von *X* hängt von der Vererbung Ihrer Klasse ab und basiert auf der Anzahl unmittelbarer Basisklassen Ihrer Klasse ab. END_RESPONSE_TABLE beendet die Definition der Ereignisantworttabelle.

Für die Definition Ihrer Antworttabelle gehen Sie wie folgt vor:

- Beginnen Sie die Definition der Antworttabelle für Ihre Klasse mit dem Makro DEFINE_RESPONSE_TABLE*X*. DEFINE_RESPONSE_TABLE*X* erwartet *X* + 1 Argumente:
  den Namen der Klasse, für die Sie die Antworttabelle definieren.
  den Namen jeder unmittelbaren Basisklasse.
- Tragen Sie die Ergebnisse in die Antworttabelle ein.
- Beenden Sie die Antworttabellendefinition mit dem Makro END_RESPONSE_TABLE.

Die Antworttabelle für *TMyFrame*, die von *TFrameWindow* abgeleitet ist, würde beispielsweise wie folgt aussehen:

```
DEFINE_RESPONSE_TABLE1(TMyFrame, TFrameWindow)
  EV_WM_LBUTTONDOWN,
  EV_WM_LBUTTONUP,
  EV_WM_MOUSEMOVE,
  EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;
```

Sie müssen hinter jedem Antworttabelleneintrag ein Komma und hinter dem Makro END_RESPONSE_TABLE ein Semikolon plazieren.

Definieren der Antworttabelleneinträge

Antworttabelleneinträge verbinden ein Windows-Ereignis mit einer speziellen Funktion. Wenn ein Fenster oder Dialogelement eine Botschaft empfängt, durchsucht es seine Antworttabelle nach einem Eintrag für diese Botschaft. Liegt ein Eintrag vor, übergibt es die Botschaft an diese Funktion. Liegt keiner vor, übergibt es die Botschaft an sein übergeordnetes Fenster. Wenn das Fenster das Hauptfenster ist, übergibt es die Botschaft an das Anwendungsobjekt. Wenn das Anwendungsobjekt keinen Antworteintrag für diese spezielle Botschaft besitzt, wird die Botschaft von der Standardverarbeitung von ObjectWindows bearbeitet. Dieser Prozeß ist in Kapitel 3 im *ObjectWindows Referenzhandbuch* detaillierter beschrieben.

ObjectWindows bietet eine große Anzahl von Makros für Antworttabelleneinträgen. Dazu gehören:

- Kommandobotschaftsmakros, die die Behandlung von Kommandobotschaften und die Zuteilung dieser Botschaften an eine spezielle Funktion ermöglichen
- Makros für Standardbotschaften von Windows für die Behandlung von Windows-Botschaften

- Registrierte Botschaften (Botschaften, die von *RegisterWindowMessage* zurückgegeben werden)
- Makros für Benachrichtigungen, die auf der ID untergeordneter Fenster basieren. Sie ermöglichen die Behandlung dieser Benachrichtigungscodes beim untergeordneten oder übergeordneten Fenster.
- Makros für Dialogelementbenachrichtigungen, um Botschaften von spezialisierten Dialogelementen wie Schaltern, Kombinationsfenster, Editierfelder, Listenfelder usw. zu behandeln
- Makros für Dokumentverwaltungsbotschaften, um die Anwendung zu benachrichtigen, daß ein Dokument oder ein View erzeugt oder entfernt wurde, und um Views über Ereignisse in der Dokumentverwaltung zu benachrichtigen.
- VBX-Dialogelementbenachrichtigungen

Kommandobotschaftsmakros

ObjectWindows bietet eine große Anzahl von Makros (Kommandobotschaftsmakros). Diese Makros können Kommandobotschaften jeder Funktion zuweisen. Die einzige Anforderung besteht darin, daß der Kopf der Funktion, die die Botschaft behandeln soll, mit dem vom Makro für diese Botschaft geforderten übereinstimmen muß. Die unterschiedlichen Typen von Kommandobotschaftsmakros sind:

Tabelle 5.1: Kommandobotschaftsmakros

Makro	Prototyp	Beschreibung
EV_COMMAND(CMD, UserName)	**void** UserName()	Ruft die Elementfunktion *UserName* auf, wenn die Kommandobotschaft CMD empfangen wurde.
EV_COMMAND_AND_ID(CMD, UserName)	**void** UserName(WPARAM)	Ruft die Elementfunktion *UserName* auf, wenn die Kommandobotschaft CMD empfangen wurde. Übergibt die Kommando-ID (der Parameter WPARAM) an die Funktion.
EV_COMMAND_ENABLE(CMD, UserName)	**void** UserName(TCommandEnabler&)	Wird zum automatischen Aktivieren und Deaktivieren von Kommandodialogelementen wie Menüeinträgen, Tool-Balken-Schalter usw. verwendet.

Es gibt andere Botschaftsmakros, um die unbehandelte Botschaft an die Funktion zur Ereignisbehandlung übergeben zu können. Diese Botschaftsmakros behandeln jede Art von generischer und registrierter Botschaft.

Tabelle 5.2: Botschaftsmakros

Makro	Prototyp	Beschreibung
EV_MESSAGE(MSG, UserName)	LRESULT *UserName*(WPARAM, LPARAM)	Ruft die Elementfunktion *UserName*, wenn die benutzerdefinierte Botschaft MSG empfangen wurde. MSG wird an *UserName* ohne Änderungen übergeben.
EV_REGISTERED(MSG, UserName)	LRESULT *UserName*(WPARAM, LPARAM)	Ruft die Elementfunktion *UserName* auf, wenn die registrierte Botschaft MSG empfangen wurde. MSG wird an *UserName* ohne Änderungen übergeben.

Es ist sehr wichtig, daß der Funktionskopf korrekt mit dem Makro, das Sie in der Definition der Antworttabelle verwenden, übereinstimmt. Nehmen wir zum Beispiel an, Sie haben den folgenden Code:

```
class TMyFrame : public TFrameWindow {
public:
  void CmAdvise();

  DECLARE_RESPONSE_TABLE(TMyFrame);
  };

DEFINE_RESPONSE_TABLE(TMyFrame, TFrameWindow)
  EV_COMMAND_AND_ID(CM_ADVISE, CmAdvise),
END_RESPONSE_TABLE;

void TMyFrame::CmAdvise()  {
  .
  .
  .
}
```

Dieser Code erzeugt zum Zeitpunkt der Compilierung einen Fehler, da das Makro EV_COMMAND_AND_ID eine Funktion erfordert, die **void** zurückgibt und als einzigen Parameter WPARAM erwartet. In diesem Beispiel gibt die Funktion zwar korrekterweise ein **void** zurück, sie erwartet aber unkorrekterweise keine Parameter. Damit dieser Code korrekt compiliert, müssen Sie einfach die Elementdeklaration und Funktionsdefinition von *TMyFrame::CmAdvise* wie folgt ändern:

```
void TMyFrame::CmAdvise(WPARAM cmd);
```

Windows-Botschaftsmakros

ObjectWindows bietet vordefinierte Makros für alle Standardbotschaften von Windows. Sie können diese Makros verwenden, um Standardbotschaften von Windows in einer der Elementfunktionen Ihrer Klasse zu behandeln.

Um den Makronamen zu ermitteln, müssen Sie dem Botschaftsnamen das Präfix »EV_« voransetzen. So hat zum Beispiel das Makro für die Be-

handlung der Botschaft WM_PAINT den Namen EV_WM_PAINT, das Makro für die Botschaft WM_LBUTTONDOWN heißt EV_WM_LBUTTONDOWN.

Diese vordefinierten Makros übergeben die Botschaft an Funktionen mit vordefiniertem Namen. Um den Funktionsnamen zu ermitteln, müssen Sie »WM_« vom Botschaftsnamen entfernen, »Ev« an den verbleibenden Namen hinzufügen und dann den Namen zu Kleinbuchstaben konvertieren, wobei an den Wortgrenzen Großbuchstaben stehen. So wird zum Beispiel die Botschaft WM_PAINT an eine Funktion mit Namen *EvPaint* übergeben, die Botschaft WM_LBUTTONDOWN an eine Funktion mit Namen *EvLButtonDown*.

Der Vorteil der Verwendung dieser vordefinierten Makros besteht darin, daß die Windows-Botschaft automatisch »extrahiert« wird. Das heißt, die normalerweise in den Parametern *LPARAM* und *WPARAM* kodierten Parameter werden in ihre Bestandteile aufgebrochen und an die Funktion zur Ereignisbehandlung als individuelle Parameter übergeben.

Das Makro EV_WM_CTLCOLOR übergibt zum Beispiel die extrahierten Parameter an eine Funktion zur Ereignisbehandlung mit folgendem Kopf:

```
HBRUSH EvCtlColor(HDC hDCChild, HWND hWndChild, UINT nCtrlType);
```

Das Extrahieren von Botschaften bietet Ihnen strikte C++ Typenprüfung zum Zeitpunkt der Compilierung. Damit werden Fehler schon beim Compilieren Ihres Codes und nicht erst zur Laufzeit abgefangen. Es hilft ebenso beim Portieren einer Anwendung von 16-Bit auf 32-Bit und umgekehrt. Das Kapitel im *ObjectWindows Referenzhandbuch* listet jede Windows-Botschaft zusammen mit ihrem entsprechendem Antworttabellenmakro und dem Kopf der entsprechenden Funktion zur Ereignisbehandlung auf.

Um ein vordefiniertes Botschaftsmakro von Windows zu verwenden, müssen Sie das Makro zu Ihrer Antworttabelle und die passende Elementfunktion mit korrektem Namen und Kopf zu Ihrer Klasse hinzufügen. Nehmen wir zum Beispiel an, Sie wollten eine Operation ausführen, wenn Ihr Fensterobjekt *TMyFrame* die Botschaft WM_ERASEBKGND empfängt. Der Code könnte in etwa wie folgt aussehen:

```
class TMyFrame : public TFrameWindow {
public:
  BOOL EvEraseBkgnd(HDC);

  DECLARE_RESPONSE_TABLE(TMyFrame);
  };

DEFINE_RESPONSE_TABLE(TMyFrame, TFrameWindow)
  EV_WM_ERASEBKGND,
END_RESPONSE_TABLE;

BOOL TMyFrame::EvEraseBkgnd(HDC hdc) {
  .
  .
  .
}
```

Benachrichtigungs-Botschaftsmakros mit der ID untergeordneter Fenster

Makros für Benachrichtigungsbotschaften, die auf der ID untergeordneter Fenster basieren, bieten eine Reihe unterschiedlicher Wege, Botschaften zu behandeln. Sie können Benachrichtigungscodes von mehreren untergeordneten Fenstern mit einer einzelnen Funktion behandeln, alle Benachrichtigungscodes eines untergeordneten Fensters an ein Antwortfenster übergeben oder den Benachrichtigungscode beim untergeordneten Fenster behandeln.

Sie verwenden diese Makros, um die Kontrolle und die Kommunikation mit untergeordneten Dialogelementen zu erleichtern. Die verschiedenen Typen von Makros für Benachrichtigungsbotschaften, die auf der ID untergeordneter Fenster basieren, sind in der folgenden Liste aufgeführt:

Tabelle 5.3: Benachrichtigungsmakros mit der ID untergeordneter Fenster

Makro	Prototyp	Beschreibung
EV_CHILD_NOTIFY(ID, Code, UserName)	**void** UserName()	Teilt die Botschaft und den Benachrichtigungscode der Elementfunktion *UserName* zu
EV_CHILD_NOTIFY_AND_CODE(Id, Code, UserName)	**void** UserName(WPARAM code)	Teilt die Botschaft *Id* mit dem Benachrichtigungscode *Code* der Funktion *UserName* zu.
EV_CHILD_NOTIFY_ALL_CODES(Id, UserName)	**void** UserName(WPARAM code)	Teilt die Botschaft *Id* der Funktion *UserName* zu, ungeachtet des Benachrichtigungscodes der Botschaft.
EV_CHILD_NOTIFY_AT_CHILD(Code, UserName)	**void** UserName()	Teilt den Benachrichtigungscode *Code* der Elementfunktion *UserName* des untergeordneten Fensterobjektes zu

Diese Makros bieten verschiedene Methoden für die Behandlung von Benachrichtigungscodes, die auf der ID untergeordneter Fenster basieren. Wenn Sie wollen, daß Benachrichtigungen mit der ID untergeordneter Fenster im übergeordneten Fenster behandelt werden, verwenden Sie EV_CHILD_NOTIFY, das den Benachrichtigungscode als ein Parameter übergibt und das die Behandlung mehrerer Benachrichtigungen mit der ID untergeordneter Fenster mit einer einzelnen Funktion erlaubt. Es verhindert ebenso, daß Sie jede Benachrichtigungsbotschaft der untergeordneten Fenster in separaten Antworttabellen für jedes Dialogelement behandeln müssen. Stattdessen wird jede Botschaft beim übergeordneten Fenster behandelt. Damit kann zum Beispiel ein Dialogfenster alle Dialogelemente in seiner Antworttabelle behandeln.

Nehmen wir zum Beispiel an, Sie haben ein Dialogfenster mit Namen *TTestDialog*, welches vier Schalter enthält. Die Schalter-IDs sind ID_BUT-

TON1, ID_BUTTON2, ID_BUTTON3 und ID_BUTTON4. Wenn der Benutzer einen Schalter anklickt, soll eine einzelne Funktion das Ereignis behandeln, unabhängig davon, welcher Schalter gedrückt wurde. Wenn der Benutzer auf einen Schalter zweimal klickt, soll eine spezielle Funktion das Ereignis behandeln. Der Code dafür würde in etwa wie folgt aussehen:

```
class TTestDialog : public TDialog {
public:
  TTestDialog(TWindow* parent, TResId resId);

  void HandleClick();
  void HandleDblClick1();
  void HandleDblClick2();
  void HandleDblClick3();
  void HandleDblClick4();

  DECLARE_RESPONSE_TABLE(TTestDialog);
};

DEFINE_RESPONSE_TABLE1(TTestDialog, TDialog)
  EV_CHILD_NOTIFY(ID_BUTTON1, BN_CLICKED, HandleClick),
  EV_CHILD_NOTIFY(ID_BUTTON2, BN_CLICKED, HandleClick),
  EV_CHILD_NOTIFY(ID_BUTTON3, BN_CLICKED, HandleClick),
  EV_CHILD_NOTIFY(ID_BUTTON4, BN_CLICKED, HandleClick),
  EV_CHILD_NOTIFY(ID_BUTTON1, BN_DOUBLECLICKED, HandleDblClick1),
  EV_CHILD_NOTIFY(ID_BUTTON2, BN_DOUBLECLICKED, HandleDblClick2),
  EV_CHILD_NOTIFY(ID_BUTTON3, BN_DOUBLECLICKED, HandleDblClick3),
  EV_CHILD_NOTIFY(ID_BUTTON4, BN_DOUBLECLICKED, HandleDblClick4),
END_RESPONSE_TABLE;
```

Wenn Sie wollen, daß alle Benachrichtigungscodes von den untergeordneten Fenstern an das übergeordnete Fenster übergeben werden, verwenden Sie EV_CHILD_NOTIFY_ALL_CODES, die generische Behandlung für Benachrichtigungen, die auf der ID untergeordneter Fenster basieren. Das Beispielprogramm BUTTONX.CPP auf Ihrer mitgelieferten Diskette definiert beispielsweise die folgende Antworttabelle:

```
DEFINE_RESPONSE_TABLE1(TTestWindow, TWindow)
  EV_COMMAND(ID_BUTTON, HandleButtonMsg),
  EV_COMMAND(ID_CHECKBOX, HandleCheckBoxMsg),

  EV_CHILD_NOTIFY_ALL_CODES(ID_GROUPBOX, HandleGroupBoxMsg),
END_RESPONSE_TABLE;
```

Diese Tabelle behandelt Botschaften von Schaltern, Markierungsfeldern und Gruppenfeldern. In diesem Fall erhält das übergeordnete Fenster (*TTestWindow*) alle Benachrichtigungsbotschaften, die vom untergeordneten Fenster (ID_GROUPBOX) gesendet wurden. Das Makro EV_CHILD_NOTIFY_ALL_CODES verwendet die benutzerdefinierte

Funktion *HandleGroupBoxMsg* für die Verarbeitung dieser Botschaften. Wenn der Benutzer mit der Maus auf eines der Schaltfelder des Gruppenfeldes klickt, erscheint als Ergebnis ein Meldungsfenster, das dem Benutzer mitteilt, welcher Schalter ausgewählt wurde.

Sie können das Makro EV_CHILD_NOTIFY_AND_CODE verwenden, wenn das übergeordnete Fenster mehr als nur eine Botschaft mit derselben Funktion behandeln soll. Ein Beispiel:

```
DEFINE_RESPONSE_TABLE1(TTestWindow, TWindow)
  EV_CHILD_NOTIFY_AND_CODE(ID_GROUPBOX, SomeNotifyCode,
HandleThisMessage),
  EV_CHILD_NOTIFY_AND_CODE(ID_GROUPBOX, AnotherNotifyCode,
HandleThisMessage),
  END_RESPONSE_TABLE;
```

Wenn Ihr Fenster mehrere verschiedene Botschaften behandeln muß und mehrere verschiedene Funktionen dazu verwendet, sollten Sie besser EV_CHILD_NOTIFY_AND_CODE an Stelle von EV_CHILD_NOTIFY benutzen, da dessen Funktion zur Botschaftsbehandlung keine Parameter empfängt und deshalb nicht weiß, welche Botschaft zu behandeln ist.

Um Benachrichtigungen mit der ID untergeordneter Fenster beim untergeordneten Fenster zu behandlen, verwenden Sie EV_CHILD_NOTIFY_AT_CHILD. Das Beispielprogramm NOTITEST.CPP auf Ihrer mitgelieferten Diskette enthält die folgende Antworttabelle:

```
DEFINE_RESPONSE_TABLE1(TBeepButton, TButton)
  EV_NOTIFY_AT_CHILD(BN_CLICKED, BnClicked),
END_RESPONSE_TABLE;
```

Diese Antworttabelle verwendet das Makro EV_NOTIFY_AT_CHILD, um dem untergeordneten Fenster (*TBeepButton*) mitzuteilen, daß der Benachrichtigungscode (BN_CLICKED) mit Hilfe der Funktion *BnClicked behandelt werden soll.*

K A P I T E L 6

Befehlsaktivierung

Dieses Kapitel behandelt die ObjectWindows Implementierung der Befehlsaktivierung. Die meisten Anwendungen verwenden Menüeinträge und Dialogelementleisten oder Palettenschalter-Gadgets, um ihre Funktionen zu erreichen. Einige der von diesen Kontrollelementen bestimmten Befehle sind nicht immer erreichbar. Die Menüeinträge und Schalter sollten dem Benutzer irgendwie anzeigen, wann der Befehl nicht erreichbar ist. Menüeinträge und Schalter-Gadgets können auch einen Anwendungsstatus mitteilen, wie zum Beispiel das eingestellte Zeichenformat oder ob eine Funktion ein- oder ausgeschaltet ist, usw.

ObjectWindows sieht dafür einen *Befehlsaktivierung* genannten Mechanismus vor, den Sie für einige wichtige Anwendungsmerkmale verwenden können. Dieses Kapitel beschreibt den Gebrauch der ObjectWindows Befehlsaktivierung für:

- Das Ein- bzw. Ausschalten der Menüauswahl und Schalter-Gadgets
- Das Setzen des Status schaltbarer Einträge, wie Menüeinträge und Schalter der Dialogelementleiste, die an- und abgeklickt werden können
- Das Ändern des Textes von Menüeinträgen

Informationen zu den Menüs finden Sie in Kapitel 9. Informationen zu den Schalter-Gadgets, wie z.B. Dialogelementleisten- oder Palettenschalter und Gadget-Fenster, wie Dialogelementleisten und Statuszeilen finden Sie in Kapitel 13.

Befehlsaktivierungsbotschaften

Die Grundidee, die hinter der ObjectWindows Befehlsaktivierung steht, ist, daß die Entscheidung, ob eine Funktion aktiviert wird oder nicht, von dem Objekt gefällt werden sollte, welches den Befehl bearbeitet. Das macht ObjectWindows, indem es eine WM_COMMAND_ENABLE Bot-

schaft in dieselbe Befehlskette schreibt wie ein WM_COMMAND Ereignis. Dieses Ereignis wird dann von dem Fensterobjekt empfangen, das genau für die Funktion, die Sie (de-)aktivieren, zuständig ist. Die Befehlsereigniskette wird in Kapitel 5 erklärt.

Wann eine WM_COMMAND_ENABLE Botschaft gesendet wird, hängt von dem Typ des Befehlseintrags ab, der hier betroffen ist. *TFrameWindow* führt die Befehlsaktivierung aus, wenn ein Benutzer ein Menü anklickt, und sendet eine WM_INITMENUPOPUP Botschaft. Gadget-Fenster führen die Befehlsaktivierung für die Schalter der Dialogelementleiste während der Idleverarbeitung eines Fensters aus.

Um Befehlsaktivierungsbotschaften für eine spezielle Funktion zu handhaben,

1. fügen Sie zur Fensterklasse eine Elementfunktion hinzu, welche die Befehlsaktivierungsbotschaft bearbeitet. Der Rückgabewert sollte **void**, der einzige Übernahmeparameter eine Referenz auf ein *TCommandEnabler* Objekt sein. Die abstrakte Basisklasse *TCommandEnabler* wird in der ObjectWindows Header-Datei window.h deklariert.
2. plazieren Sie das EV_COMMAND_ENABLE Makro in die Antworttabelle des übergeordneten Fensters. Dieses Makro übernimmt zwei Parameter, den Befehlsbezeichner und den Namen der Behandlungsfunktion.

Betrachten Sie z.B. eine Fensterrahmenklasse, die einen Befehl *Datei/Sichern* verarbeitet, der wiederum den Befehlbezeichner CM_FILESAVE verwendet. Die Klassendefinition würde ungefähr so aussehen:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow *parent = 0, char *title = 0)
      : TFrameWindow(parent, title), IsDirty(false) {}

  protected:
    void CmFileSave();

  DECLARE_RESPONSE_TABLE(TMyFrame);
};

DEFINE_RESPONSE_TABLE(TMyFrame)
```

```
  EV_COMMAND(CM_FILESAVE, CmFileSave),
END_RESPONSE_TABLE;
```

Angenommen Sie möchten nicht, daß der Benutzer den Befehl *Datei/Sichern* ausführen kann, weil er die Datei seit ihrem Öffnen oder letzten Sichern noch nicht verändert hatte. Das Hinzufügen einer Behandlungsfunktion und eines Antworttabellenmakros, um die *CmFileSave* Funktion zu aktivieren, sieht etwa so aus:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow *parent = 0, char *title = 0)
      : TFrameWindow(parent, title), IsDirty(false) {}

  protected:
    void CmFileSave();

    // Das ist die Behandlungsfunktion der Befehlsaktivierung.
    void CeFileSave(TCommandEnabler& commandEnabler);

  DECLARE_RESPONSE_TABLE(TMyFrame);
};

DEFINE_RESPONSE_TABLE(TMyFrame)
  EV_COMMAND(CM_FILESAVE, CmFileSave),
  EV_COMMAND_ENABLE(CM_FILESAVE, CeFileSave),
END_RESPONSE_TABLE;
```

Beachten Sie, daß das EV_COMMAND Makro und das EV_COMMAND_ENABLE Makro beide denselben Befehlsbezeichner verwenden. Oft kann eine einzelne Funktion auf verschiedene Weise angesprochen werden. Viele Anwendungen öffnen z.B. eine Datei mittels eines Menüeintrags, aber auch mittels eines Schalters auf der Dialogelementleiste. Befehlsaktivierung in ObjectWindows erlaubt Ihnen, Befehle auf jede Weise zu aktivieren, indem Sie eine Funktion durch einen einzigen gemeinsamen Bezeichner aufrufen. Durch die Abstrahierung der Befehlsaktivierung über Befehlsaktivierungsobjekte sparen Sie viel Zeit, da Sie nicht für jeden unterschiedlichen Befehlseintrag Befehlsaktivierungsfunktionen schreiben müssen.

Arbeiten mit Befehlsaktivierungsobjekten

Sind Sie einmal so weit, daß eine Befehlsaktivierungsbotschaft empfangen wurde und die zugehörige Behandlungsfunktion aufgerufen wurde, können Sie verschiedene Aktionen durch Übergabe des Befehlsaktivierungsobjekts an die Behandlungsfunktion einleiten. Dieser Abschnitt erläutert die verschiedenen Typen von ObjectWindows Befehlsaktivierungsobjekten.

ObjectWindows Befehlsaktivierungsobjekte

ObjectWindows enthält drei vordefinierte Befehlsaktivierungsobjekte:

TCommandEnabler ist die abstrakte Basisklasse für Befehlsaktivierungsobjekte. Deklariert ist es in der ObjectWindows Header-Datei window.h.

TMenuItemEnabler ist die Befehlsaktivierungsklasse für Menüeinträge. Diese Klasse (de-)aktiviert Menüeinträge, plaziert Checkmarken in und ändert den Text von Menüeinträgen. Diese Klasse wird in der ObjectWindows Quelldatei FRAMEWIN.CPP deklariert.

TButtonGadgetEnabler ist die Befehlsaktivierungsklasse für Schalter-Gadgets. Diese Klasse (de-)aktiviert Schalter-Gadgets und schaltet boolesche Schalter Gadgets um. Diese Klasse wird in der ObjectWindows Quelldatei BUTTONGA.CPP deklariert.

TCommandEnabler: die Befehlsaktivierungsschnittstelle

Obwohl Sie in Ihren Befehlsaktivierungsfunktionen immer ein von *TCommandEnabler* abgeleitetes Objekt, im Gegensatz zu einem direkten *TCommandEnabler* Objekt, manipulieren, erscheint es aber doch in der Praxis so, als ob Sie mit einem *TCommandEnabler* Objekt arbeiten. *TCommandEnabler* enthält eine zusammenhängende Schnittstelle für die anderen Befehlsaktivierungsklassen, die eine ähnliche Funktionalität für den Typ des Befehlsobjekts implementiert, den jede Klasse liefert. Weil Sie niemals eine Instanz der *TMenuItemEnabler* und *TButtonGadgetEnabler* Klassen erzeugen werden, sind diese in Quelldateien anstatt in Header-Dateien deklariert. Es ist nicht notwendig, daß Sie eines dieser Objekte erzeugen können; statt dessen arbeiten Sie mit der Basis-*TCommandEnabler* Schnittstelle, während Ihre Behandlungsfunktionen das spezielle Befehlswerkzeug, das gerade benutzt wird, ignorieren.

Dieser Abschnitt behandelt die *TCommandEnabler* Funktionsschnittstelle. Es gibt zwei Methoden für die *TCommandEnabler* Funktionsschnittstelle:

- Wenn Sie vorhandene Befehlsaktivierungsklassen verwenden, müssen Sie sich mit der Basisschnittstelle, wie sie in der *TCommandEnabler* Klasse implementiert ist, auskennen.
- Wenn Sie neue Befehlsaktivierungsklassen ableiten, müssen Sie sich mit der aktuellen Implementierung der Funktionalität in der *TCommandEnabler* Basisklasse auskennen.

Dieser Abschnitt behandelt beide Methoden und legt dar, welche Aspekte für den Gebrauch existierender Klassen und zur Erzeugung neuer Klassen relevant sind.

Funktionen

TCommandEnabler besitzt eine Anzahl von Elementfunktionen:

- Der Konstruktor von *TCommandEnabler* ist nur dann interessant, wenn Sie eine neue Befehlsaktivierungsklasse ableiten - die Klasse ist abstrakt. Er übernimmt zwei Parameter, ein uint und ein HWND. Der uint ist der Befehlsbezeichner. Der Konstruktor initialisiert das *Id* Datenelement mit dem Wert des Befehlsbezeichners. Der HWND ist die Kennung für das Fenster, das die Befehlsaktivierungsbotschaft erhält. Der Konstruktor initialisiert *HWndReceiver* mit dem Wert des HWND Parameters.
- *Enable* übernimmt einen einzelnen booleschen Parameter und übergibt **void**. Der boolesche Parameter zeigt an, ob der Befehl (de-)aktiviert werden soll; im Falle von true aktiv, von false inaktiv.

Von der Ableitung neuer Klassen ausgehend führt der *TCommandEnabler::Enable* eine Initialisierung der Datenelemente in der Basisklasse aus. Alle anderen Aktionen zum (De-)Aktivieren von Befehlseinträgen müssen in der abgeleiteten Klasse stehen. Beispielsweise führt *TMenuItemEnabler* alle notwendigen Arbeiten aus, um Menüeinträge ein- oder auszuschalten. *Enable* Funktionen von abgeleiteten Klassen sollten immer *TCommandEnabler::Enable* aufrufen.

- *SetText* übernimmt einen einzelnen Parameter, einen **const char far***, und gibt **void** zurück. Diese Funktion setzt den Text des Befehlseintrags in den String ein, der an den Zeichen-Array Parameter übergeben wird. *SetText* wirkt nicht auf Schalter-Gadgets.

SetText ist rein virtuell deklariert; Sie *müssen SetText* in Klassen, die von *TCommandEnabler* abgeleitet sind, deklarieren und definieren. Jeder Schritt, der benötigt wird, um diese Funktionalität in Ihren Befehlseintrag zu implementieren, muß in die abgeleitete *SetText* Funktion eingetragen

werden. Wenn es, wie das in *TButtonGadgetEnabler* der Fall ist, keine gültige Anwendung für die *SetText* Funktion gibt, können Sie sie einfach als Leerfunktion implementieren.

- *SetCheck* übernimmt einen einzelnen **int** Parameter und gibt **void** zurück. Diese Funktion schaltet den Befehlseintrag ein oder aus, abhängig von dem Wert des **int** Parameters. Dieser Paramaeter kann einer von drei Aufzählungswerten sein, die in der *TCommandEnabler* Klasse definiert wurden, *Unchecked*, *Checked* oder *Indeterminate*. *Unchecked* setzt den Status des Befehlseintrags auf ungeprüft, *Checked* setzt den Status des Befehlseintrags auf geprüft, und *Indeterminate* setzt den Befehlseintrag in den Unbestimmtheitszustand. Dieser Unbestimmtheitszustand wird über den Befehlseintrag definiert:
- Für Menüeintrage ist der Unbestimmtheitszustand gleich dem ungeprüften Status.
- Für Schalter-Gadgets ist der Unbestimmtheitszustand ein Status zwischen geprüft und ungeprüft.

SetCheck ist rein virtuell deklariert; Sie *müssen SetCheck* in Klassen, die von *TCommandEnabler* abgeleitet sind, deklarieren und definieren. Jeder Schritt, der benötigt wird, um diese Funktionalität in Ihren Befehlseintrag zu implementieren, muß in die abgeleitete *SetCheck* Funktion eingetragen werden.

- *GetHandled* benötigt keine Parameter und gibt bool zurück. Diese Funktion liefert true zurück, wenn der Befehlsaktivierer über einen Aufruf der *Enable* Funktion gestartet worden ist. Andernfalls liefert sie false.
- *IsReceiver* übernimmt einen einzelnen HWND Parameter und liefert einen booleschen Wert. *IsReceiver* gibt true zurück, wenn der HWND Parameter dem Empfänger HWND gleicht, der dem *TCommandEnabler* Konstruktor übergeben und in *HWndReceiver* abgelegt wurde. Andernfalls liefert sie false.

Datenelemente

TCommandEnabler besitzt drei Datenelemente:

- *Id* ist das einzige Public-Datenelement. Diese Variable enthält den Befehlsbezeichner. Sie ist deklariert worden als **const** uint und wird durch den Konstruktor initialisiert. Einmal initialisiert, kann sie nicht mehr verändert werden.
- *HWndReceiver* hält das Handle des Fensters, das den Befehl implementiert. Dieses Datenelement ist protected und kann nicht direkt verändert werden, außer wenn Sie es aus der Klasse *TCommandEnabler* ableiten. Auf *HWndReceiver* kann indirekt, durch Aufruf der *IsReceiver* Funk-

tion, die den Wert des übergebenen HWND Parameters mit dem Wert von *HWndReceiver* vergleicht, zugegriffen werden.

- *Handled* gibt an, ob sich das Befehlsaktivierungsobjekt mit etwas befaßt hat. Initialisiert mit false durch den *TCommandEnabler* Konstruktor, wird es auf true gesetzt durch *TCommandEnabler::Enable*. Dieses Datenelement ist protected und kann nicht direkt verändert werden, außer wenn Sie es aus der Klasse *TCommandEnabler* ableiten. Auf *Handled* kann indirekt mittels Aufruf der *GetHandled* Funktion, die den Wert von *Handled* liefert, zugegriffen werden.

Gewöhnliche Befehlsaktivierungstasks

Dieser Abschnitt beschreibt, wie sie mehr oder weniger gewöhnliche Tasks, für die Sie Befehlsaktivierung benutzen, erstellen können, einschließlich

- Aktivierung und Deaktivierung von Befehlseinträgen
- Ändern des Menüeintragstextes
- Umschalten von Befehlseinträgen

Aktivierung und Deaktivierung von Befehlseinträgen

(De-)Aktivierung von Befehlseinträgen ist so einfach wie ein Aufruf der *Enable* Funktion in Ihrer Behandlungsfunktion. Sie bestimmen die Kriterien, nach denen ein bestimmter Eintrag aktiviert oder deaktiviert wird. Wenn z.B. eine spezielle Bibliothek nicht vorhanden ist, möchten Sie vielleicht die Befehle deaktivieren, die versuchen, auf diese zuzugreifen. Wenn Ihre Anwendung Dateien verschiedenen Formats akzeptiert, möchten Sie vielleicht, daß die Befehle nicht aktiv sind, die dem aktuellen Format nicht entsprechen.

Um einen Befehl zu (de-)aktivieren,

1. Fügen Sie die Behandlungsfunktion der Befehlsaktivierung und das Antworttabellenmakro in Ihre Fensterklasse ein.
2. Definieren Sie die Behandlungsfunktion.
3. Rufen Sie innerhalb der Behandlungsfunktion die *Enable* Elementfunktion, die zu dem Befehlsobjekt gehört, das Sie der Behandlungsfunktion übergaben, auf. Die *Enable* Funktion übernimmt einen einzelnen booleschen Parameter. Ein Aufruf von *Enable* mit dem Parameterwert true aktiviert, bzw. false deaktiviert den Befehl.

Hier ist noch einmal die vorher behandelte Beispielklasse , zusätzlich mit einem Flag im booleschen Format, *IsDirty*, das signalisieren soll, ob die Datei, seit man sie letztmalig geöffnet oder gespeichert hat, verändert wurde; weiterhin noch mit der Funktion *CeFileSave*, die für die (De-)Aktivierung des Befehls *Datei/Sichern* zuständig ist:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow *parent = 0, char *title = 0)
      : TFrameWindow(parent, title), IsDirty(false) {}

  protected:
    bool IsDirty;

    void CmFileSave();

    // Das ist die Behandlungsfunktion der Befehlsaktivierung.
    void CeFileSave(TCommandEnabler& commandEnabler);

  DECLARE_RESPONSE_TABLE(TMyFrame);
};

DEFINE_RESPONSE_TABLE(TMyFrame)
  EV_COMMAND(CM_FILESAVE, CmFileSave),
  EV_COMMAND_ENABLE(CM_FILESAVE, CeFileSave),
END_RESPONSE_TABLE;

void
TMyFrame::CeFileSave(TCommandEnabler& ce)
{
  ce.Enable(IsDirty);
}
```

CeFileSave überprüft das *IsDirty* Flag. Falls *IsDirty* false ist (die Datei ist nicht verändert worden), deaktivieren Sie den *CmFileSave* Befehl durch Aufruf von *Enable* mit dem Parameter false. Falls *IsDirty* true ist (die Da-

tei ist verändert worden), aktivieren Sie den *CmFileSave* Befehl durch Aufruf von *Enable* mit dem Parameter true. Da Sie *Enable* mit demselben Wert wie *IsDirty* aufrufen möchten, können Sie *IsDirty* einfach als Parameter an *Enable* übergeben.

Diese Methode einen Befehl zu (de-)aktivieren, funktioniert sowohl für Menüeinträge als auch für Schalter-Gadgets. Im vorhergehenden Beispiel, wo Sie einen Schalter eines Dialogelements und einen Menüeintrag haben, die den CM_FILESAVE Befehl senden, sind beide Befehle in der *CmFileSave* Funktion implementiert. Ähnlich ist die Befehlsaktivierung für den Kontrolleistenschalter und den Menüeintrag in der *CeFileSave* Funktion implementiert.

Text des Menüeintrags verändern

Veränderung des Textes eines Menüeintrags erfolgt mit Hilfe der *SetText* Funktion. Um den Text eines Menüeintrags zu verändern:

1. Fügen Sie die Behandlungsfunktion der Befehlsaktivierung und das Antworttabellenmakro zu Ihrer Fensterklasse hinzu.
2. Definieren Sie die Behandlungsfunktion.
3. Rufen Sie in der Behandlungsfunktion die *SetText* Elementfunktion des Befehlaktivierungsobjekts auf, das an die Behandlungsfunktion übergeben wurde. *SetText* übernimmt einen einzigen Parameter, einen **const far char***. Dieser Zeichen-Array-Parameter sollte den neuen Text für den Menüeintrag beinhalten. *SetText* gibt **void** zurück.

Anmerkung

Wenn Sie den Text für einen Menüeintrag festlegen und eine Checkmarke für diesen Menüeintrag in derselben Funktion vorsehen, müssen Sie *SetText* vor *SetCheck* aufrufen. Die Umkehrung dieser Reihenfolge entfernt diese Checkmarke.

Angenommen Ihre Anwendung unterstützt drei verschiedene Dateiformate: Text, binäres und verschlüsseltes Format. Sie möchten, daß der Menüeintrag *Datei/Sichern* das Format der Datei wiedergibt, die gespeichert wird. Hier ist wieder die zuvor erwähnte Beispielklasse, modifiziert durch einen **enum**-Typ, *TFormat*, und ein *TFormat* Datenelement genannt *Format*:

```
class TMyFrame : public TFrameWindow
{
  public:
    TMyFrame(TWindow *parent = 0, char *title = 0);
    enum TFormat {Text, Binary, Encrypted};
```

```
  protected:
    TFormat Format;
    void CmFileSave();
    // Das ist die Behandlungsfunktion der Befehlsaktivierung.
    void CeFileSave(TCommandEnabler& commandEnabler);

  DECLARE_RESPONSE_TABLE(TMyFrame);
};

DEFINE_RESPONSE_TABLE(TMyFrame)
  EV_COMMAND(CM_FILESAVE, CmFileSave),
  EV_COMMAND_ENABLE(CM_FILESAVE, CeFileSave),
END_RESPONSE_TABLE;

void
TMyFrame::CeFileSave(TCommandEnabler& ce)
{
  switch(Format) {
    case Text:
      ce.SetText("Save as text file");
      break;
    case Binary:
      ce.SetText("Save as binary file");
      break;
    case Encrypted:
      ce.SetText("Save as encrypted file");
      break;
    default:
      ce.SetText("Save");
  }
}
```

Umschalten von Befehlseinträgen

Sie können das Umschalten von Befehlseinträgen verwenden, um den Benutzern Ihrer Anwendungen visuell darauf hinzuweisen, welche Funktionen aktiviert sind, die verschiedenen Anwendungszustände anzuzeigen usw. Alles, was in einer booleschen Weise dargestellt werden kann, wie etwa an und aus, hinein und hinaus usw., kann durch Umschalten von Befehlseinträgen dargestellt werden.

Es gibt zwei verschiedene Arten des Umschaltens, die in ObjectWindows implementiert sind, aber beide sind auf die gleiche Weise implementiert. Sie können Checkmarken mit Menüeinträgen ein- und ausschalten. Sie können »check« und »uncheck« auch auf Schalter-Gadgets anwenden, so daß das Gadget hervorsteht, wenn es aus ist und eingedrückt und hell ist, wenn es an ist. Es gibt auch einen dritten *indeterminate* Zustand, der anzeigt, wenn etwas weder ein- noch ausgeschaltet ist. Die Bedeutung dieses Zustandes ist im wesentlichen Ihnen überlassen, doch zeigt er gewöhnlich eine Situation an, in der die Kriterien für (De-)Aktivierung gemischt sind. Beispielsweise haben viele Textverarbeitungsprogramme Schalter für Dialogelementleisten, die das gegebene Textformat anzeigen, wie etwa einen Schalter mit der Aufschrift »F«, um fettgedruckten Text anzuzeigen. Dieser Schalter ist nicht aktiviert, wenn das gegebene Textformat nicht fett ist, und aktiviert, wenn es sich um Fettdruck handelt. Wenn aber ein Textblock Text enthält, der zum Teil fettgedruckt ist, zum Teil nicht, ist der Schalter in seiner Zwischenstellung. Abbildung 6.1 zeigt ein Schalter-Gadget in jeder der drei Stellungen:

Schalter-Gadget Status

Eine Abwandlung vom Umschalten von Schalter-Gadgets ist das (De-)Aktvieren eines exklusiven Schalter-Gadgets. Exklusive Schalter-Gadgets funktionieren genau wie ein Schaltfeld. Von einer Gruppe von exklusiven Schalter-Gadgets kann gleichzeitig immer nur ein Schalter-Gadget an sein. Aktivieren eines anderen Schalter-Gadgets in der Gruppe deaktiviert das zuvor aktivierte Schalter-Gadget.

Um einen Befehlseintrag umzuschalten,

1. Fügen Sie die Behandlungsfunktion der Befehlsaktivierung und das Antworttabellenmakro zu Ihrer Fensterklasse hinzu.
2. Definieren Sie die Behandlungsfunktion.
3. Rufen Sie in der Behandlungsfunktion die *SetCheck* Elementfunktion des Befehlsaktivierungsobjekts auf, das an die Behandlungsfunktion übergeben wurde. *SetCheck* übernimmt einen einzelnen **int** Parameter. Rufen Sie *SetCheck* mit einem der in *TCommandEnabler* definierten Aufzählungswerten auf: *Checked*, *Unchecked* oder *Indeterminate*.

Anmerkung

Wenn Sie den Text für einen Menüeintrag festlegen und eine Checkmarke für diesen Menüeintrag in derselben Funktion vorsehen, müssen Sie *SetText* vor *SetCheck* aufrufen. Die Umkehrung dieser Reihenfolge entfernt diese Checkmarke. Siehe weiter vorne für Informationen über das Setzen von Checkmarken für Menüeinträge.

Gewöhnlich bestimmt der Benutzer einer Anwendung, ob eine Eigenschaft aktiviert werden soll. Wenn beispielsweise Ihre Anwendung für den Zugriff auf die *CmFileSave* Funktion sowohl einen Menüeintrag als auch einen Schalter einer Dialogelementleiste vorsieht. Viele Anwendungen enthalten dafür sogenannte »fly-over«-Hilfen, kurze Beschreibungen, die in der Statuszeile erscheinen, wenn der Mauszeiger über einen Menüeintrag oder einen Schalter-Gadget wandert. Eventuell soll der Benutzer diese Hilfen ein- und ausschalten können. Um diese Option dem Benutzer zu überlassen:

1. Fügen Sie Ihrer Anwendung einen neuen Befehlsbezeichner hinzu, z.B. CM_TOGGLEHINTS.
2. Fügen Sie ein neues Menü hinzu, beispielsweise namens Optionen, mit einem Menüeintrag Fly-over-Hilfe.
3. Sie können auch einen neuen Schalter auf Ihre Schaltleiste setzen (siehe Kapitel 13 zur Information zum Hinzufügen neuer Schalter-Gadgets in Ihre Kontrolleiste).
4. Schreiben Sie eine Funktion, die das CM_TOGGLEHINTS Ereignis verarbeitet und damit die Hilfen tatsächlich ein- und ausschaltet.
5. Fügen Sie eine Befehlsaktivierungsfunktion hinzu, die die Befehlseinträge gegebenenfalls aktiviert.

Hier ist wieder die zuvor erwähnte Beispielklasse, modifiziert zum Gebrauch eines sichtbaren Rahmenfensters. Der Benutzer kann die Hilfen ein- und ausschalten, indem er den dafür vorgesehenen Befehlseintrag verwendet.

```
class TMyDecFrame : public TDecoratedFrame
{
  public:
    TMyDecFrame(TWindow *parent = 0, char *title = 0, TWindow* client)
      : TDecoratedFrame(parent, title, client), hintMode (true) {}

    // Cb muß vom Anwendungsobjekt über die InitMainWindow Funktion
       bestimmt werden.
    TControlBar* Cb;
```

```
  protected:
    // HintMode signalisiert ob die Hilfen ein- oder ausgeschaltet sind.
    bool HintMode;

    // Dies ist die Funktion, die tatsächlich die Hilfen ein- und
       ausschaltet.
    void CmToggleHints();

    // Das ist die Behandlungsfunktion der Befehlsaktivierung.
    void CeToggleHints(TCommandEnabler& commandEnabler);

  DECLARE_RESPONSE_TABLE(TMyDecFrame);
};

DEFINE_RESPONSE_TABLE(TMyDecFrame)
  EV_COMMAND(CM_TOGGLEHINTS, CmToggleHints),
  EV_COMMAND_ENABLE(CM_TOGGLEHINTS, CeToggleHints),
END_RESPONSE_TABLE;

void
TMyDecFrame::CmToggleHints()
{
  if(HintMode)
    Cb->SetHintMode(TGadgetWindow::EnterHints);
  else
    Cb->SetHintMode(TGadgetWindow::NoHints);
  HintMode = !HintMode;
}

void
TMyDecFrame::CeToggleHints(TCommandEnabler& ce)
{
  ce.SetChecked(HintMode);
}
```

Beachten Sie, daß die Dialogelementleiste vom Anwendungsobjekt in seiner Funktion *InitMainWindow* eingerichtet wird. Der Code dafür ist hier nicht abgebildet. Erklärungen der Anwendungsobjekte und der *InitMainWindow* Funktion finden Sie in Kapitel 3. Erklärungen der Schalter-Gadgets und Dialogelementleisten finden Sie in Kapitel 13. Arbeitsvorlagen zum Umschalten von Befehlseinträgen, finden Sie im Beispiel EXAMPLES/OWL/OWLAPPS/MDIFILE.

K A P I T E L 7

ObjectWindows Exception Handling

ObjectWindows enthält für das Auftreten von Exceptions einen stabilen Exception Handling-Mechanismus. Eine Exception (Ausnahmezustand) ist eine Situation, die in einer Anwendung auftreten kann, wenn sie außerhalb ihrer Normalparameter läuft. Das kann sowohl etwas Harmloses, wie eine unerwartete Benutzerreaktion, als auch etwas Ernsteres, wie ein ungültiges Handle oder gar ein Speicherzuweisungsfehler, sein. Exception Handling ist ein sauberer, effektiver Weg, um mit all diesen und weiteren Bedingungen fertig zu werden.

Dieses Kapitel beschreibt die ObjectWindows Einkapselung des Exception Handling, einschließlich

- Exception-Klassenhierarchie
- Exception-Ressourcen-Bezeichner
- Code-Makros, die es Ihnen erleichtern, Exception Handling ein- und auszuschalten

Sie sollten sich möglichst gut mit dem C++ Exception Handling auskennen, bevor Sie dieses Kapitel lesen. C++ Exception Handling wird beschrieben im Kapitel 4 des *Borland C++ Programmierhandbuchs*.

ObjectWindows Exception-Hierarchie

ObjectWindows enthält eine Anzahl von Klassen, die in Ausnahmesituationen Exceptions behandeln können. Basierend auf den Klassen *TXBase* und *TXOwl*, können diese Exception-Klassen den Benutzer über den momentanen Ausnahmezustand informieren, ihn zu einer Eingabe zur Bestimmung der erforderlichen Maßnahmen auffordern, neue Ausnahmeobjekte bilden, Ausnahmesituationen erzeugen, usw. Vier Exception-Klassen sind als unabhängige Klassen implementiert:

- *TXBase* ist die Basisklasse für alle ObjectWindows und ObjectComponents Exception-Klassen. *TXBase* ist von der Borland C++ Klasse *xmsg* abgeleitet. *xmsg* ist im Kapitel 4 des *Borland C++ Programmierhandbuchs* beschrieben.
- *TXOwl* wird von *TXBase* abgeleitet. *TXOwl* ist die Basisklasse für die ObjectWindows Exception-Klassen.
- *TXCompatibility* beschreibt Exceptions, die dann auftreten, wenn *TModule::Status* ungleich Null ist. Dies erhält die Abwärtskompatibilität zwischen der ObjectWindows 1.0 Methode des Aufspürens von Ausnahmesituationen und der ObjectWindows 2.*x* Exception-Hierarchie. *TXCompatibility* weist den Wert von *TModule::Status* einem Ressourcen-Zeichenkettenbezeichner zu.
- *TXOutOfMemory* beschreibt eine Exception, die immer dann auftritt, wenn ein Versuch, Speicherraum zu reservieren, fehlschlägt. Ein Analogon stellt das *xalloc* Objekt dar, wenn **new** nicht korrekt Speicher reservieren konnte.

Zwei weitere Klassen, *TXOle* und *TXAuto*, sind von *TXBase* abgeleitet. Diese Klassen enthalten das Exception Handling für die ObjectComponents Klassen. Sie werden im *ObjectWindows Referenzhandbuch* beschrieben.

Arbeiten mit TXBase

Als Basisklasse für die ObjectWindows Exception-Klassen enthält *TXBase* die Basisschnittstelle, um mit ObjectWindows Exceptions zusammenzuarbeiten. *TXBase* enthält folgende Funktionen:

- Es kann sich selbst aufbauen und sein Basisobjekt *xmsg* initialisieren.
- Es kann sich selbst klonen, indem es eine Kopie des Exception-Objektes herstellt.
- Es kann sich, wie ein Exception-Objekt, selbst auslösen.

Auf- und Abbau von TXBase

TXBase besitzt zwei öffentliche Konstruktoren:

```
TXBase(const string& msg);

TXBase(const TXBase& src);
```

Der erste Konstruktor initialisiert die Basisklasse *xmsg* mit dem Wert des Parameters *string*, wobei der Konstruktor *xmsg* so aufrufen wird, daß er den Parameter *string* übernimmt. Der zweite erzeugt ein neues Objekt,

welches eine Kopie vom Objekt *TXBase* ist, die als Parameter übergeben wird.

Beide Konstruktoren inkrementieren das Datenelement *InstanceCount* von *TXBase*. *InstanceCount* ist ein **static int**, d.h. daß es nur eine einzige Instanz dieses Elements gibt, egal wie viele *TXBase* oder von *TXBase* abgeleitete Objekte in der Anwendung vorhanden sind. Der Destruktor von *TXBase* dekrementiert *InstanceCount*. Der Destruktor ist als virtuell deklariert, damit er einfach überschrieben werden kann.

Weil jedes neue *TXBase* oder von *TXBase* abgeleitete Objekt *InstanceCount* inkrementiert, und jedes gelöschte *TXBase* oder von *TXBase* abgeleitete Objekt *InstanceCount* dekrementiert, gibt der Wert von *InstanceCount* die Gesamtzahl der *TXBase* und von *TXBase* abgeleiteten Objekte, die derzeit in der Anwendung vorkommen, wieder. Um auf *InstanceCount* von außerhalb einer *TXBase* oder von *TXBase* abgeleiteten Klasse zuzugreifen, muß vor den Namen *InstanceCount* der *TXBase::* Gültigkeitsbereichsqualifizierer gesetzt werden.

Klonen von Exception-Objekten

TXBase enthält eine Funktion namens *Clone*. Diese Funktion hat keine Parameter und liefert einen Zeiger *TXBase**. *Clone* erstellt durch Zuweisung eines neuen *TXBase* Objekts mit **new** und durch Übergabe eines dereferenzierten **this** Zeigers dem Kopierkonstruktor eine Kopie des aktuellen Exception-Objekts.

```
TXBase*
TXBase::Clone()
{
  return new TXBase(*this);
}
```

Es ist wichtig, zu beachten, daß jede Klasse, die von *TXBase* abgeleitet wurde, diese Funktion überschreiben muß, damit der richtige Konstruktor benutzt wird. Beispielsweise implementiert die von *TXBase* abgeleitete Klasse *TXOwl* die *Clone* Funktion so:

```
TXOwl*
TXOwl::Clone()
{
  return new TXOwl(*this);
}
```

Auslösung von TXBase Exceptions

Haben Sie einmal ein *TXBase* Objekt instantiiert, entweder durch Erzeugung oder durch Klonen, können Sie das Objekt auf dreierlei Weise auslösen.

- Benutzen Sie das Schlüsselwort **throw** gefolgt vom Objektnamen

```
TXBase xobj("Some exception...");

throw xobj;
```

- Verwenden Sie das ObjectWindows THROW Makro, das mit dem C++ Schlüsselwort **throw** korrespondiert. Das obige Beispiel würde dann so aussehen:

```
TXBase xobj("Some exception...");

throw xobj;
```

- Rufen Sie die *Throw* Funktion des Exception-Objekts:

```
TXBase xobj("Some exception...");

xobj.Throw();
```

Diese Methode sorgt für strenge Typsicherung beim Auslösen der Exception. Sie enthält auch eine polymorphe Schnittstelle beim Auslösen der Exception, so daß die Funktion, die ein von TXBase abgeleitetes Exception-Objekt empfängt, dieses wie *TXBase* selbst behandeln kann, egal woraus es besteht.

Arbeiten mit TXOwl

Als Basisklasse für ObjectWindows Ausnahmeklassen enthält *TXOwl* die Basisschnittstelle für das Arbeiten mit den ObjectWindows Exceptions. Zusätzlich zu der Funktionalität, die in der Klasse *TXBase* enthalten ist, kann *TXOwl* noch weitere Funktionen ausführen.

- Es kann sich selbst aufbauen und seine Basisobjekte initialisieren.
- Es kann sich selbst klonen, indem es eine Kopie des Exceptionobjekts herstellt.
- Es kann unbehandelte Exceptions der Funktion *Error* des Anwendungsobjekts oder dem globalen Exception Handler *HandleGlobalException* (*HandleGlobalException* wird später diskutiert) übergeben.

Auf- und Abbau von TXOwl

TXOwl besitzt zwei Konstruktoren, die für eine flexible Übergabe der Zeichenkette für die Ausnahmemeldung sorgen:

```
TXOwl(const string& str, unsigned resId = 0);

TXOwl(unsigned resId, TModule* module = ::Module);
```

Der erste Konstruktor initialisiert das *TXBase* Basisobjekt mit dem Wert des *string* Parameters. Der **unsigned** Parameter wird als Errornummer verwendet.

Der zweite Konstruktor lädt die String-Ressource, identifiziert über *resId*, und verwendet die Zeichenkette, um *TXBase* zu initialisieren. Der Zeiger *TModule** identifiziert das Modul, von welchem die Ressource geladen werden sollte. Er zeigt standardmäßig auf den momentanen globalen Modulzeiger *Module*, d.h. daß die Resource vom momentanen Modul oder der Anwendung geladen werden sollte.

Der *TXOwl* Destruktor führt standardmäßig nur die aus *TXBase* vererbten Funktionen aus.

Klonen von TXOwl und von TXOwl abgeleiteten Exception-Objekten

TXOwl enthält auch die *Clone* Funktion. Diese Funktion hat keine Parameter und liefert einen Zeiger *TXOwl**. *Clone* erstellt durch Zuweisung eines neuen *TXOwl* Objektes mit **new** und durch Übergabe eines dereferenzierten **this** Zeigers an den automatischen Kopierkonstruktor eine Kopie des aktuellen Exception-Objekts.

```
TXOwl*

TXOwl::Clone()

{

  return new TXOwl(*this);

}
```

Es ist wichtig, zu beachten, daß jede Klasse, die von *TXOwl* abgeleitet wurde, diese Funktion überschreiben muß, damit der richtige Konstruktor benutzt wird. Beispielsweise implementiert die von *TXOwl* abgeleitete Klasse *TXOutOfMemory* die *Clone* Funktion so:

```
TXOwl*

TXOutOfMemory::Clone()

{
```

```
  return new TXOutOfMemory(*this);
}
```

Beachten Sie, daß der Typ des Rückgabewerts weiterhin *TXOwl** ist. Dies ermöglicht den ObjectWindows Exception Handling-Funktionen, jedes Exception-Objekt wie ein *TXOwl* Objekt zu behandeln, und somit die polymorphe Eigenschaft der ObjectWindows Hierarchie zu erhalten. Beachten Sie jedoch auch, daß sich der Typ des Rückgabewertes für *TXOwl::Clone* von dem der *TXBase::Clone* Funktion unterscheidet. Und zwar deswegen, weil *TXBase* die Basisfunktionalität für die Exception-Klassen von ObjectWindows und ObjectComponents gewährleistet, wohingegen *TXOwl* die Basisschnittstelle für die ObjectWindows Exception-Klassen enthält.

Spezialisierte ObjectWindows Exception-Klassen

Einige gewöhnliche ObjectWindows Klassen implementieren spezialisierte Exception-Klassen; jede davon basiert auf *TXOwl*, ist aber definiert innerhalb der implementierenden Klassendefinition, um den Gültigkeitsbereich ihrer Namen gemäß einzurichten. Die folgende Tabelle enthält diese Klassen und ihre spezielle Funktion. Die verschiedenen IDS_* Ressourcen, die in der Tabelle erwähnt sind, werden mit vielen weiteren in Kapitel 2 des *ObjectWindows Referenzhandbuchs* vorgestellt.

Tabelle 7.1: Spezialisierte Exception-Klassen

Elternklasse	Exception-Klasse	Funktion
TApplication	*TXInvalidMainWindow*	Initialisiert die Ausnahmemeldung mit der IDS_INVALIDMAINWINDOW String-Ressource. Dieses Objekt wird ausgelöst, wenn das *MainWindow* Element von *TApplication* entweder einen ungültigen Zeiger oder einen Zeiger auf ein ungültiges Fenster enthält.
TModule	*TXInvalidModule*	Initialisiert die Ausnahme-meldung mit der IDS_INVALIDMODULE String-Ressource. Diese Exception wird im Konstruktor *TModule* ausgelöst, wenn *HInstance* des Moduls ungültig ist.
Twindow	*TXWindow*	Initialisiert die Ausnahmemeldung mit dem Fenstertitel und einer String-Ressource, die dem *TXWindow* Konstruktor übergeben wird. Diese Exception wird in Situationen ausgelöst, wenn ein Fehler bezüglich eines Fensterobjekts aufgetreten ist.
TXmenu	*TXMenu*	Initialisiert die Ausnahmemeldung mit einer String-Ressource, die dem *TXMenu* Konstruktor

		übergeben wird. Das ist standardmäßig die IDS_GDIFAILURE String-Ressource. Diese Exception wird ausgelöst, wenn ein Menüobjekt-Handle ungültig ist.
TValidator	*TXValidator*	Initialisiert die Ausnahmemeldung mit einer String-Ressource, die dem *TXValidator* Konstruktor übergeben wird. Das ist standardmäßig die IDS_VALIDATORSYNTAX String-Ressource. Diese Exception wird ausgelöst, wenn eine Prüfungsausnahme zerstört oder ungültig ist.
TGdiBase	*TXGdi*	Initialisiert die Ausnahmemeldung mit einer String-Ressource, die dem *TXGdi* Konstruktor übergeben wird, zusammen mit dem GDI-Objekt-Handle. Standardmäßig besitzt die String-Ressource den Wert IDS_GDIFAILURE und das GDI Objekt-Handle ist 0. Diese Exception wird in vielen Situationen ausgelöst, wenn ein Fehler durch ein Grafikobjekt entsteht.
TPrinter	*TXPrinter*	Initialisiert die Ausnahmemeldung mit einer String-Ressource, die dem *TXPrinter* Konstruktor übergeben wird. Das ist standardmäßig die IDS_PRINTERERROR String-Ressource. Diese Exception wird ausgelöst, wenn der Gerätekontext des Druckers ungültig ist.

ObjectWindows Exception Handling-Makros

ObjectWindows enthält einige Makros zum implementieren des Exception Handling. Obwohl Sie weiterhin die Standard C++ Schlüsselwörter wie **try**, **catch**, **throw** usw. benutzen können, befähigen Sie die ObjectWindows Makros dazu, die Exception Handling einfach durch die Definition eines einzelnen Symbols ein- und auszuschalten. Es gibt folgende Makros

- TRY
- THROW(x)
- THROWX(x)
- RETHROW
- CATCH(x)

Diese Makros werden in diesem Abschnitt weiter hinten erklärt.

ObjectWindows Exceptions ein- und ausschalten

Das Symbol, das die Exception Handling in ObjectWindows ein- und ausschaltet, ist NO_CPP_EXCEPTIONS. Der Wert (oder fehlende Wert), der NO_CPP_EXCEPTIONS zugewiesen ist, spielt keine Rolle. Relevant ist nur, ob er definiert ist. Falls nicht, werden die Exception Handling-Makros expandiert, um das Exception Handling einzubauen. Falls doch, stellen die Makros nur die Mindestfunktionalität ein, das Beenden der Anwendung, wenn eine Exception ausgelöst wird. Die genauen Verfahrensweisen der Makros, wenn das Exception Handling ein- und ausgeschaltet wird, werden später beschrieben.

Es gibt viele verschiedene Methoden, um NO_CPP_EXCEPTIONS zu definieren. Diese Liste enthält nicht alle Möglichkeiten seiner Definition, doch schlägt sie einige vor.

- Sie können die -DNO_CPP_EXCEPTIONS Option auf der MAKE Kommandozeile bestimmen. So wird das Makro definiert, jedoch mit keinem spezifischen Wert.
- Sie können ein Symbol definieren, indem Sie eine grafische Entwicklungsumgebung, wie etwa die Borland C++ IDE, benutzen. Verwenden Sie die Methode, die in Ihrer grafischen Entwicklung zur Verfügung steht, um das NO_CPP_EXCEPTIONS Symbol zu definieren.
- Sie können NO_CPP_EXCEPTIONS in Ihrem Quelltext definieren. Dies ist eine weniger geeignete Methode als die vorgenannten, hauptsächlich weil die Erzeugung Ihrer Anwendung mit MAKE oder anderen Abhängigkeiten überprüfenden Programmen den Zeitstempel Ihrer Datei verändert, wenn Sie den Quelltext verändern. Das hängt davon ab, ob Sie den Zeitstempel ändern wollen oder nicht.

Makroexpansion

Die Exception Handling-Makros in ObjectWindows verhalten sich unterschiedlich, je nachdem ob NO_CPP_EXCEPTIONS definiert ist oder nicht. Die folgende Tabelle führt aus, wie jedes Makro expandiert wird, abhängig vom Status von NO_CPP_EXCEPTIONS:

Tabelle 7.2: ObjectWindows Exception Handling-Makroexpansion

Makro	NO_CPP_EXCEPTIONS definiert	NO_CPP_EXCEPTIONS nicht definiert
TRY	Nicht expandierend, löscht die Anweisung **try**.	Expandiert zu **try**, erlaubt dem Code im **try** Block, auf ausgelöste Exceptions getestet zu werden.

THROW(x)	Ruft die Funktion *abort* auf.	Expandiert zu **throw**(*x*), löst das *x* Objekt aus, wenn die Ausnahmebedingungen zutreffen.
THROWX(x)	Ruft die Funktion *abort* auf.	Expandiert zu *x.Throw()*, ruft die *Throw* Function des Objekts *x*. Dieses Makro sollte nur von *TXBase* abgeleiteten Klassen verwendet werden.
RETHROW	Nicht expandierend, löscht die Anweisung **throw**.	Expandiert zu **throw**. Dieses Makro sollte nur innerhalb von **catch** (oder CATCH) verwendet werden, welches es durch Wiederauslösen der erfaßten Exception abschließt.
CATCH(x)	Nicht expandierend, löscht die Anweisung **catch**.	Expandiert zu **catch** x, erfaßt Exceptions ausgelöst von Objekten des Typs *x*.

K A P I T E L 8

Fensterobjekte

Fensterobjekte sind hochentwickelte Oberflächenelemente, die die Kontrolle der Fenster, der untergeordneten Fenster und der Dialogelemente ermöglichen. ObjectWindows bietet mehrere verschiedene Typen von Fensterobjekten:

- Layoutfenster
- Rahmenfenster
- Dekorierte Rahmenfenster
- MDI-Fenster

Es existiert noch eine weitere Klasse von Fensterobjekten, die Gadget-Fenster genannt werden. Diese Objekte werden in Kapitel 13 beschrieben.

Verwenden von Fensterobjekten

In diesem Abschnitt wird erläutert, wie Fensterobjekte erzeugt, angezeigt und gefüllt werden. Die Ausführung folgender Aufgaben wird beschrieben:

- Konstruieren eines Fensterobjektes
- Einstellen von Attributen für die Erzeugung
- Erzeugen von Oberflächenelementen für Fenster

Die in diesem Kapitel beschriebenen verschiedenen Fenstertypen – Rahmenfenster, Layoutfenster, dekorierte Rahmenfenster und MDI-Fenster – sind Beispiele für Fensterobjekte. Die Information in diesem Abschnitt bezieht sich auf sämtliche Typen von Fensterobjekten.

Konstruieren von Fensterobjekten

Fensterobjekte repräsentieren Oberflächenelemente. Das Objekt ist mit dem Element über ein Handle verbunden, welches im Datenelement *HWindow* des Objektes gespeichert ist. *HWindow* ist von *TWindow* geerbt. Wenn Sie ein Fensterobjekt konstruieren, existiert sein Oberflächenelement noch nicht. Sie müssen es in einem extra Schritt erzeugen. *TWindow* besitzt darüber hinaus einen Konstruktor, den Sie in einer DLL verwenden können, um ein Fensterobjekt für ein bereits existierendes Oberflächenelement zu erzeugen.

Eine Reihe von ObjectWindows 2.5-Klassen sind von *TWindow* und *TFrameWindow* abgeleitet, wobei diese die Basisklassen als virtuelle Basisklassen benutzen. Diese Klassen heißen *TDialog*, *TMDIFrame*, *TTinyCaption*, *TMDIChild*, *TDecoratedFrame*, *TLayoutWindow*, *TClipboardViewer*, *TKeyboardModeTracker* und *TFrameWindow* (von der *TWindow* als virtuelle Basisklasse benutzt wird). In C++ werden virtuelle Basisklassen zuerst konstruiert. Das bedeutet, daß der Konstruktor der abgeleiteten Klasse keine Standardargumente für den Basisklassen-Konstruktor angeben kann. Für dieses Problem gibt es zwei Lösungsmöglichkeiten:

- Konstruieren Sie Ihre unmittelbare Basisklasse oder Klassen und alle virtuellen Basisklassen, wenn Sie Ihre abgeleiteten Klassen konstruieren. Ihre ObjectWindows 1.0-Code könnte zum Beispiel wie folgt aussehen:

```
class TMyWindow : public TWindow
{
  public:
    TMyWindow(PTWindowsObject parent, LPSTR title)
      : TWindow(parent, title) {}
};
```

Unter ObjectWindows 2.5 würde dieser Code wie folgt aussehen:

```
class TMyWindow : public TFrameWindow
{
  public:
    TMyWindow(TWindow *parent, const char *title)
      : TFrameWindow(parent, title),
        TWindow(parent, title) {}
};
```

- Verwenden Sie den Standardkonstruktor der virtuellen Basisklasse. Sowohl *TWindow* als auch *TFrameWindow* besitzen einen Standardkonstruktor. Zur Angabe von Parametern für die Basisklassen besitzen diese beiden Klassen zudem eine Funktion *Init*. Sie sollten diese *Init*-Funktionen im Konstruktor Ihrer abgeleiteten Klasse aufrufen, um alle gewünschten Parameter in der Basisklasse zu setzen.

Wenn wir mit dem ObjectWindows 1.0-Code aus dem obigen Beispiel starten, würde der ObjectWindows 2.5-Code in etwa wie folgt aussehen:

```
class TMyWindow : public TFrameWindow
{
  public:
    TMyWindow(TWindow *parent, const char *title) {
      Init();
    }
};
```

Im Folgenden werden einige Beispiele für das Konstruieren von Fensterobjekten mit den oben beschriebenen Methoden angeführt:

```
TWindow *myWindow1 = new TWindow(this, "A window's title";
TFrameWindow myWindow2(0,"My window's title", new TMDIClient, TRUE);

class TMyWin : public TFrameWindow
{
  public:
    TMyWin(TWindow *parent, char *title)
      : TFrameWindow(parent, title),
        TWindow(parent, title) {}
}

TMyWin *myWin = new TMyWin(GetMainWindow(), "Child window");

class TNewWin : virtual public TWindow

{
  public:
    TNewWin(TWindow *parent, char *title);
}

TNewWin::TNewWin(TWindow *parent, char *title)
{
  Init(parent, title, IDL_DEFAULT);
};

TNewWin *newWin = new TMyWin(GetMainWindow(), "Child window");
```

Einstellen von Attributen für die Erzeugung

Eine typische Windows-Anwendung hat viele verschiedene Fenstertypen: überlappende Fenster oder Popup-Fenster, Fenster mit Rahmen, Fenster mit Bildlaufleiste oder Fenster mit Titel (um nur einige zu nennen). Die verschiedenen Typen werden mit *Eigenschaftsattributen* kontrolliert. Eigenschaftsattribute werden wie der Titel eines Fensters während der Initialisierung eines Fensterobjektes gesetzt und während der Erzeugung des Oberflächenelementes benutzt.

Die Attribute eines Fensterobjektes zur Erzeugung, wie Eigenschaften und Titel, werden im Element *Attr* des Objektes gespeichert. Dieses Element ist eine *TWindowAttr*-Struktur. Die folgende Tabelle zeigt die Elemente von *TWindowAttr*. Die darunterliegende Abbildung zeigt, wie einige dieser Attribute das Aussehen des Fensters beeinflussen.

Tabelle 8.1: Attribute für die Erzeugung von Fenstern

Element	Typ	Beschreibung
Style	*DWORD*	Eigenschaftskonstante.
ExStyle	*DWORD*	Erweiterte Eigenschaftskonstante.
X	**int**	Die horizontale Bildschirmkoordinate der linken oberen Fensterecke
Y	**int**	Die vertikale Bildschirmkoordinate der linken oberen Fensterecke
W	**int**	Die Anfangsbreite des Fensters in Bildschirmkoordinaten
H	**int**	Die Anfangshöhe des Fensters in Bildschirmkoordinaten
Menu	*TResId*	ID der Menüressource des Fensters. Sie sollten nicht versuchen, einen Menübezeichner direkt *Attr.Menu* zuzuweisen! Verwenden Sie stattdessen die Funktion *AssignMenu*.
Id	**int**	Id des untergeordneten Fensters für die Kommunikation zwischen einem Dialogelement und seinem übergeordneten Fenster. *ID* sollte für alle untergeordneten Fenster eines übergeordneten Fensters eindeutig sein. Wenn das Dialogelement in einer Ressource definiert ist, sollte seine *Id* dieselbe wie die der Ressourcen-ID sein. Ein Fenster sollte niemals gleichzeitig *Menu* und *ID* gesetzt haben.
Param	**char far ***	Wird bei *TMDIClient* benutzt, um Informationen über den MDI-Rahmen und untergeordnete Fenster festzuhalten.
AccelTable	*TResId*	ID der Ressource für die Tastenkürzeltabelle des Fensters

Überschreiben von Standardattributen.

Die Tabelle listet die Standardattribute für die Erzeugung auf. Sie können diese Vorgabewerte im Konstruktor einer abgeleiteten Fensterklasse durch die Änderung der Werte in der *Attr*-Struktur überschreiben, wie zum Beispiel in:

Listing 8.1: Überschreiben von Standardattributen in einem Fensterkonstruktor

```
TTestWindow::TTestWindow(TWindow* parent, const char* title)
  : TFrameWindow(parent, title),
    TWindow(parent, title)
{
  Attr.Style &= ~(WS_SYSMENU | WS_MAXIMIZEBOX);
  Attr.Style |= WS_MINIMIZEBOX;
  Attr.X = 100;
  Attr.Y = 100;
  Attr.W = 415;
  Attr.H = 355;
}
```

Attribute untergeordneter Fenster

Um die Attribute eines untergeordneten Fensters zu verändern, können Sie den Konstruktor des untergeordneten wie auch den des übergeordneten Fensterobjektes benutzen. Wenn Sie die Attribute im Konstruktor des

übergeordneten Fensterobjektes ändern, benötigen Sie den Zeiger zum untergeordneten Fensterobjekt, um auf dessen *Attr*-Element zugreifen zu können.

Listing 8.2: Überschreiben von Attributen eines untergeordneten Fensters im Konstruktor des übergeordneten Fensters

```
TTestWindow::TTestWindow(TWindow* parent, const char* title)
  : TWindow(parent, title)
{
  TWindow helpWindow (this, "Help System");

  helpWindow.Attr.Style |= WS_POPUPWINDOW | WS_CAPTION;
  helpWindow.Attr.X = 100;
  helpWindow.Attr.Y = 100;
  helpWindow.Attr.W = 300;
  helpWindow.Attr.H = 300;
  helpWindow.SetCursor(0, IDC_HAND);
}
```

Die folgende Tabelle zeigt einige Vorgabewerte, die Sie bei *Attr*-Elementen eventuell überschreiben müssen:

Tabelle 8.2: Standardattribute für Fenster

Ein Vorgabewert von 0 bedeutet, daß der Windows-Vorgabewert verwendet werden muß.

***Attr*-Element**	**Vorgabewert**
Style	WS_CHILD | WS_VISIBLE
ExStyle	0
X	0
Y	0
W	0
H	0
Menu	0
Id	0
Param	0
AccelTable	0

Erzeugen von Oberflächenelementen für Fenster

Wenn Sie ein Fensterobjekt konstruiert haben, müssen Sie Windows auffordern, das zugehörige Oberflächenelement zu erzeugen. Rufen Sie dazu die Elementfunktion *Create* des Objektes auf:

```
window.Create();
```

Create führt die folgenden Aktionen aus:

- Erzeugt das Oberflächenelement
- Setzt *HWindow* auf das Handle des Oberflächenelements
- Setzt Elemente von *Attr* auf den aktuellen Zustand des Oberflächenelements (*Style*, *ExStyle*, *X*, *Y*, *H*, *W*)
- Ruft *SetupWindow* auf

Während der Erzeugung eines Oberflächenelementes zu einem Fensterobjekt können zwei C++ Exceptions auftreten. *Create* ruft eine *TXInvalidWindow*-Exception hervor, wenn das Fenster nicht erzeugt werden kann. *SetupWindow* ruft *TXInvalidChildWindow* hervor, wenn in einem Fenster das untergeordnete Fenster nicht erzeugt werden kann. Beide Exceptions werden normalerweise durch unzureichenden Speicher oder andere Ressourcen verursacht. Sie sollten daher die Aufrufe von *Create* in einem **try**/**catch**-Block einschließen, um die in Ihrer Anwendung eventuell auftretenden Speicher- oder Ressourceprobleme abzufangen.

Das Hauptfenster einer Anwendung wird durch *TApplication::InitInstance* automatisch erzeugt. Sie müssen *Create* nicht selbst aufrufen, um das Hauptfenster zu erzeugen.

Layoutfenster

Dieser Abschnitt beschreibt Layoutfenster. Layoutfenster sind in der Klasse *TLayoutWindow* eingebettet, die von *TWindow* abgeleitet ist. Zusammen mit *TFrameWindow* bietet *TLayoutWindow* die Basis für dekorierte Rahmenfenster und deren Fähigkeit, Dekorationen im Rahmenbereich anzuordnen.

Layoutfenster besitzen diesen Namen, da sie untergeordnete Fenster im Client-Bereich des Layoutfensters anordnen. Die Positionen der untergeordneten Fenster werden in bezug auf das Layoutfenster oder auf ein anderes untergeordnetes Fenster (als **gleichgeornetes** Fenster bekannt) gesehen. Die Position eines untergeordneten Fensters hängt von den *Layoutmetriken* dieses Fensters ab. Layoutmetriken bestehen aus einer Anzahl von Regeln, die die X- und Y-Koordinaten des Fensters, dessen Höhe und dessen Breite beschreiben. Diese Regeln basieren normalerweise auf den Koordinaten eines gleichgeordneten Fensters und letzten Endes auf der Größe und der Anordnung des Layoutfensters.

Layoutmetriken für ein untergeordnetes Fenster sind in einer Klasse mit Namen *TLayoutMetrics* enthalten. Ein Layoutmetrikobjekt besteht aus einer Reihe von *Layoutbedingungen*. Jede Layoutbedingung beschreibt eine Regel für das Ermitteln einer bestimmten "Dimension", wie zum Beispiel die X-Koordinate oder die Breite des Fensters. Man benötigt vier Layoutbedingungen, um ein Layoutmetrikobjekt vollständig zu beschreiben. Layoutbedingungen sind in einer Struktur mit Namen *TLayoutConstraints* enthalten. Normalerweise werden Sie jedoch eine der von *TLayoutConstraints* abgeleiteten Klassen, wie zum Beispiel *TEdgeConstraint*, *TEdgeOrWidthConstraint* oder *TEdgeOrHeightConstraint*, verwenden.

Die nächsten Abschnitte beschreiben die Verwendung von Layoutfenstern. Zuerst werden die Layoutbedingungen erläutert. Danach wird die Definition der Layoutmetrik durch die Layoutbedingungen beschrieben. Im letzten Abschnitt wird auf die Verwendung von Layoutmetrikobjekten mit untergeordneten Fenstern in einem Layoutfenster eingegangen.

Layoutbedingungen

Layoutbedingungen geben das Verhältnis zwischen der Kante oder Dimension des einen Fensters und der Kante oder Dimension eines gleichgeordneten Fensters an. Ebenso können die Bedingungen auch das Verhältnis eines untergeorneten Fenster zu seinem übergeordneten Layoutfenster beschreiben. Sie können dieses Verhältnis auf mehrere Arten definieren. So können Sie zum Beispiel die Breite eines Fensters im Verhältnis zur Breite des übergeordneten Fensters festlegen. Damit wird bei jeder Veränderung der Größe des übergeordneten Fensters auch die Größe des untergeordneten Fensters geändert. So bleibt der vom untergeordneten Fenster beanspruchte Fensterbereich im Verhältnis gleich. Sie können auch die linke Kante eines Fensters mit der rechten Kante eines anderen untergeordneten Fensters gleichsetzen. Damit bleiben die beiden Fenster miteinander verbunden, wenn sie verschoben werden. Sie können ferner für ein Fenster festlegen, daß es eine absolute Größe und Position im Client-Bereich belegt.

Die drei am häufigsten benutzten Typen von Bedingungen sind *TEdgeConstraint*, *TEdgeOrWidthConstraint* und *TEdgeOrHeightConstraint*. Diese Strukturen enthalten alle Bedingungen, die in der Klasse *TLayoutMetrics* benutzt werden. *TEdgeOrWidthConstraint* und *TEdgeOrHeightConstraint* sind von *TEdgeConstraint* abgeleitet. Oberflächlich betrachtet, sehen diese Objekte fast identisch aus. Wenn in diesem Abschnitt von *TEdgeConstraint* die Rede ist, werden damit die Klassen *TEdgeConstraint*, *TEdgeOrWidthConstraint* und *TEdgeOrHeightConstraint* gemeint; es sei denn, die anderen Klassen werden bei der Erklärung explizit ausgeschlossen.

Definieren von Bedingungen

Grundsätzlich erfolgt die Definition der auf Bedingungen basierenden Verhältnisse – d.h. die Definition eines Verhältnisses zwischen einer Kante oder Größe des einen Fensters und einer Kante oder Größe eines anderen Fensters – über die Funktion *Set*. Diese Funktion ist in der Klasse *TEdgeConstraint* definiert und damit wird sie in die Klassen *TEdgeOrWidthConstraint* und *TEdgeOrHeightConstraint* vererbt.

Hier ist nun die Funktionsdeklaration von *Set*:

```
void Set(TEdge edge, TRelationship rel,
         TWindow* otherWin, TEdge otherEdge,
         int value = 0);
```

Dabei gilt Folgendes:

- *edge* gibt an, welcher Teil des Fensters einer Bedingung unterliegt. Dazu existiert der Aufzählungstyp **enum***TEdge*. *TEdge* besitzt fünf mögliche Werte:

  lmLeft gibt die linke Fensterkante an.

  lmTop gibt die obere Fensterkante an.

  lmRight gibt die rechte Fensterkante an.

  lmBottom gibt die untere Fensterkante an.

  lmCenter gibt die Fenstermitte an. Das Objekt, welches die Bedingung besitzt (z.B. *TLayoutMetrics*), entscheidet, ob dies die vertikale oder horizontale Mitte bedeutet.

Zusätzlich können Sie für die Breite oder Höhe eines Fensters einer Bedingung festlegen. Dies ist allerdings nur mit den Klassen *TEdgeOrWidthConstraint* und *TEdgeOrHeightConstraint* möglich. Hierzu existiert das **enum** TWidthHeight. *TWidthHeight* besitzt zwei mögliche Werte:

lmWidth gibt an, daß für die Breite des Fensters eine Bedingung definiert werden soll.

lmHeight gibt an, daß für die Höhe des Fensters eine Bedingung definiert werden soll.

- *rel* gibt das Verhältnis zwischen zwei Kanten an:

rel	**Verhältnis**
lmAsIs	Diese Dimension wird auf ihre aktuelle Größe festgelegt.
lmPercentOf	Diese Dimension wird auf ein bestimmtes Verhältnis zur Größe der bedingenden Kante festgelegt. Dies wird üblicherweise mit einer Breite oder Höhe gekoppelt.
lmAbove	Diese Dimension wird auf einen bestimmten Abstand oberhalb ihrer bedingenden Kante festgelegt.
lmLeftOf	Diese Dimension wird auf einen bestimmten Abstand links von ihrer bedingenden Kante festgelegt.
lmBelow	Diese Dimension wird auf einen bestimmten Abstand unterhalb ihrer bedingenden Kante festgelegt.
lmRightOf	Diese Dimension wird auf einen bestimmten Abstand rechts von ihrer bedingenden Kante festgelegt.
lmSameAs	Diese Dimension wird auf denselben Wert wie ihre bedingende Kante festgelegt.
lmAbsolute	Diese Dimension wird auf eine absolute Koordinate oder Größe festgelegt.

- *otherWin* gibt das Fenster an, nach dem sich die Bedingung Ihres untergeordneten Fensters richten soll. Sie müssen den Wert *lmParent* verwenden, wenn Sie das übergeordnete Fenster angeben.

- otherEdge gibt eine einzelne Kante von *otherWin* an, nach der sich die Bedingung Ihres untergeordneten Fensters richten soll. *otherEdge* kann einen der auch für *edge* erlaubten Werte annehmen.
- value hat in Abhängigkeit des Wertes von *rel* unterschiedliche Bedeutungen:

rel	**Bedeutung von *value***
lmAsIs	*value* hat keine Bedeutung und sollte auf Null gesetzt sein.
lmPercentOfvalue	zeigt an, um welchen Prozentbetrag sich die beiden Kanten unterscheiden.
lmAbove	*value* zeigt an, wieviele Einheiten die eine (bedingte) Kante oberhalb der anderen (bedingende) Kante liegen soll.
lmLeftOf	*value* zeigt an, wieviele Einheiten die eine (bedingte) Kante links von der anderen (bedingende) Kante liegen soll.
lmBelow	*value* zeigt an, wieviele Einheiten die eine (bedingte) Kante unterhalb der anderen (bedingenden) Kante liegen soll.
lmRightOf	*value* zeigt an, wieviele Einheiten die eine (bedingte) Kante rechts von der anderen (bedingenden) Kante liegen soll.
lmSameAs	*value* hat keine Bedeutung und sollte auf Null gesetzt sein.
lmAbsolute	*value* ist das absolute Maß für die bedingte Kante: Wenn *edge* den Wert *lmLeft*, *lmRight* oder *lmCenter* hat, ist *value* die X-Koordinate für die Kante. Wenn *edge* den Wert *lmTop*, *lmBottom* oder *lmCenter* hat, ist *value* die Y-Koordinate für die Kante. Wenn *edge* den Wert *lmWidth* oder *lmHeight* hat, repräsentiert *edge* die Länge des Bedingungungsparameters. Ob *lmCenter* eine X-Koordinate oder eine Y-Koordinate repräsentiert, hängt vom besitzenden Objekt ab.

Die Bedeutung von *value* hängt ebenso vom Wert von *Units* ab. *Units* ist ein *TMeasurementUnits*-Element von *TLayoutConstraint*. *TMeasurementUnits* ist ein **enum**, das den von *value* repräsentierten Einheitentyp beschreibt. *Units* kann entweder *lmPixels* oder *lmLayoutUnits* sein. *lmPixels* zeigt an, daß *value* eine absolute Zahl physikalischer Pixel repräsentieren soll. *lmLayoutUnits* zeigt an, daß *value* eine Anzahl logischer Einheiten repräsentieren soll. Diese Layouteinheiten basieren auf der Größe der aktuellen Schrift des Layoutfensters.

TEdgeConstraint enthält ferner eine Anzahl von Funktionen, die Sie für die Einstellung vordefinierter Verhältnisse benutzen können. Diese korrespondieren eng mit den Verhältnissen, die Sie in der Funktion *Set* angeben können. In Wirklichkeit rufen diese Funktionen *Set* auf, um das auf Bedingungen basierende Verhältnis zu definieren. Sie können diese Funktionen verwenden, um einen Großteil Ihrer auf Bedingungen basierenden Verhältnisse zu definieren.

Die folgenden vier Funktionen arbeiten sehr ähnlich:

```
void LeftOf(TWindow* sibling, int margin = 0);
void RightOf(TWindow* sibling, int margin = 0);
void Above(TWindow* sibling, int margin = 0);
void Below(TWindow* sibling, int margin = 0);
```

Jede dieser Funktionen plazieren das untergeordnete Fenster in ein bestimmtes Verhältnis zum bedingenden Fenster *sibling*. Die Kanten sind vordefiniert, wobei die bedingte Kante auf der Gegenseite der im Funktionsnamen beschriebenen liegt und die bedingende Kante mit dem Funktionsnamen übereinstimmt.

Die Funktion *LeftOf* plaziert zum Beispiel das untergeordnete Fenster links von *sibling*. Das bedeutet, daß die bedingte Kante des untergeordneten Fensters *lmRight* ist und die bedingende Kante von *sibling lmLeft*.

Mit der Funktion *Absolute* können Sie eine Kante Ihres untergeordneten Fensters auf einen absoluten Wert einstellen:

```
void Absolute(TEdge edge, int value);
```

edge zeigt an, welche Kante einer Bedingung unterliegen soll, und *value* hat den gleichen Wert wie bei *Set* mit dem Verhältnis *lmAbsolute*.

Es existieren noch zwei weitere abkürzende Funktionen:

```
void SameAs(TWindow* otherWin, TEdge edge);
void PercentOf(TWindow* otherWin, TEdge edge, int percent);
```

Diese beiden Funktionen beziehen sich auf dieselbe Kante für das bedingte und das bedingende Fenster. Das heißt, wenn Sie *lmLeft* für *edge* angeben, richtet sich die linke Kante Ihres untergeordneten (bedingten) Fensters nach der linken Kante von *otherWin* (bedingendes Fenster).

Definieren bedingenden Verhältnissen

Eine einzelne Layoutbedingung reicht für die Anordnung eines Fensters nicht aus. Wenn Sie zum Beispiel angeben, daß ein Fenster zehn Pixel unterhalb eines anderen Fensters liegen soll, wird keine Aussage über die Breite oder Höhe, die Positionen der linken und unteren Ränder oder über die Position des unteren Randes getroffen. Es wird damit nur festgelegt, daß eine Kante zehn Pixel unterhalb eines anderen Fensters positioniert wird.

Mit einer Kombination von Layoutbedingungen können Sie die Position eines Fensters vollständig festlegen. Die Klasse *TLayoutMetrics* verwendet vier Strukturen für Layoutbedingungen – zwei *TEdgeConstraint*-Objekte mit den Namen *X* und *Y*, ein *TEdgeOrWidthConstraint* mit Namen *Width* und ein *TEdgeOrHeightConstraint* mit Namen *Height*.

TLayoutMetrics ist eine einfache Klasse. Der Konstruktor erwartet keine Parameter. Die einzige von ihm ausgeführte Aktion besteht darin, daß je-

des Layoutbedingungselement initialisiert wird. Dazu wird Folgendes durchgeführt:

- Das bedingende Fenster wird auf Null gesetzt.
- Das Verhältnis wird auf *lmAsIs* gesetzt.
- Die Einheiten werden auf *lmLayoutUnits* gesetzt.
- der Wert wird auf Null gesetzt.

Der einzige Unterschied besteht bei *MyEdge*. *MyEdge* gibt die Kante an, auf die diese Bedingung wirkt. *X* wird auf *lmLeft*, *Y* auf *lmTop*, *Width* auf *lmWidth* und *Height* auf *lmHeight* gesetzt.

Wenn Sie ein *TLayoutMetrics*-Objekt konstruiert haben, müssen Sie die Layoutbedingungen für das anzuordnende Fenster festlegen. Sie können die im vorigen Abschnitt beschriebenen Funktionen für das Festlegen jeder Layoutbedingung verwenden.

Sie sollten beachten, daß die Namen *X*, *Y*, *Width* und *Height* der Einfachheit halber vergeben wurden und weniger strikte Regeln beschreiben, wie die Bedingungen verwendet werden sollen. *X* kann die X-Koordinate der linken Kante, der rechten Kante oder der Mitte repräsentieren. Sie können dies mit der *Width*-Bedingung kombinieren, die *lmCenter*, *lmRight* oder *lmWidth* annehmen kann, um Position und Breite des Fensters auf der X-Achse vollständig zu definieren. Eine Verwendung aller Kantenbedingungen ist dann sinnvoll, wenn Fenster nebeneinander angeordnet werden.

Die einfachste Möglichkeit besteht in der Zuweisung einer X-Koordinate an *X* und einer Breite an *Width*. Sie könnten aber ebenso die Kante für *X* auf *lmCenter* und die Kante für *Width* auf *lmRight* setzen. Damit repräsentiert *Width* keine wirkliche Breite, sondern die X-Koordinate der rechten Fensterkante. Wenn Sie die X-Koordinate der rechten Kante und der Mitte kennen, kann die X-Koordinate der linken Kante auf einfache Weise berechnet werden.

Für ein besseres Verständnis davon, wie Bedingungen zur Beschreibung eines Fensters zusammenarbeiten, sollten Sie die Beispielanwendung LAYOUT im Verzeichnis EXAMPLES\OWL\OWLAPI\LAYOUT erzeugen und starten. Diese Anwendung besitzt eine Reihe von untergeordneten Fenstern in einem Layoutfenster. Über das Menü können Sie auf ein Dialogfenster zugreifen. Dieses Dialogfenster ermöglicht Ihnen die Änderung von Bedingungen für jedes Fenster. Sie können sich dann als Ergebnis die neuen Fensteranordnungen ansehen. Seien Sie aber trotzdem vorsichtig: Wenn Sie eine Anzahl von Layoutbedingungen angeben, die ein Fenster nicht vollständig beschreiben, wird die Anwendung eventuell abstürzen oder es wird eine Überprüfung stattfinden, sofern die Diagnose eingeschaltet ist. Der Grund hierfür wird im folgenden Abschnitt erläutert.

Unvollständige Bedingungen

Bei der Angabe Ihrer Layoutbedingungen sollten Sie sorgfältig vorgehen. Die in der Klasse *TLayoutMetrics* verfügbaren Bedingungen ermöglichen Ihnen zwar eine vollständige Beschreibung eines Fensters. Sie garantieren jedoch nicht, daß die von Ihnen verwendeten Bedingungen ein Fenster auch vollständig beschreiben. In den Fällen, in denen eine Beschreibung unvollständig ist, wird ein Absturz der Anwendung die wahrscheinlichste Folge sein.

Verwenden von Layoutfenstern

Sobald Sie die Layoutbedingungen eingerichtet haben, können Sie ein Layoutfenster erzeugen, in dem die untergeordneten Fenster angeordnet werden. Dies ist der Konstruktor für *TLayoutWindow*:

```
TLayoutWindow(TWindow*        parent,
              const char far* title = 0,
              TModule*        module = 0);
```

Dabei gilt:

- *parent* ist das dem Layoutfenster übergeordnete Fenster.
- title ist der Titel des Layoutfensters. Dieser Parameter enthält als Vorgabewert einen leeren String.
- module wird als *TModule*-Parameter an den Konstruktor der Basisklasse *TWindow* übergeben. Dieser Parameter hat standardmäßig den Wert Null.

Nach dem Konstruieren und Anzeigen des Layoutfensters steht Ihnen eine Reihe von Funktionen für einen Aufruf zur Verfügung:

- Die Funktion *Layout* erwartet keine Parameter und gibt **void** zurück. Sie teilt dem Layoutfenster mit, alle seine untergeordneten Fenster neu anzuordnen. Mit diesem Aufruf zwingen Sie das Fenster, die Grenzen und Positionen jedes untergeordneten Fensters neu zu berechnen. Ein Aufruf von *Layout* wird üblicherweise dann erforderlich, wenn ein untergeordnetes Fenster bewegt wurde, die Größe des Layoutfensters verändert wurde oder eine andere Aktion stattfand, die Einfluß auf die Bedingungen der untergeordneten Fenster haben könnte.

Beachten Sie, daß *TLayoutWindow* die *TWindow*-Version von *EvSize* überschreibt, um *Layout* immer dann automatisch aufzurufen, wenn ein WM_SIZE-Ereignis abgefangen wurde. Wenn Sie diese Funktion selbst überschreiben, sollten Sie sicherstellen, daß die Basisklassenversion dieser Funktion aufgerufen wird oder aber *Layout* in Ihrer abgeleiteten Version aufgerufen wird.

- *SetChildLayoutMetrics* erwartet ein *TWindow* **&** und ein *TLayoutMetrics* **&** als Parameter und gibt **void** zurück. Verwenden Sie diese Funktion, um eine Menge von Bedingungen, die in einem *TLayoutMetrics*-Objekt enthalten ist, auf ein untergeordnetes Fenster anzuwenden. Es folgt ein Bei-

spiel für die Erzeugung eines *TLayoutMetrics*-Objektes und dessen Anwendung auf ein untergeordnetes Fenster:

```
TMyLayoutWindow::TMyLayoutWindow(TWindow* parent, char far* title)
  : TLayoutWindow(parent, title)
{
  TWindow MyChildWindow(this);

  TLayoutMetrics layoutMetrics;

  layoutMetrics.X.Absolute(lmLeft, 10);
  layoutMetrics.Y.Absolute(lmTop, 10);
  layoutMetrics.Width.PercentOf(lmParent, lmWidth, 60);
  layoutMetrics.Height.PercentOf(lmParent, lmHeight, 60);

  SetChildLayoutMetrics(MyChildWindow, layoutMetrics);
}
```

Beachten Sie, daß das untergeordnete Fenster für die Assoziation mit dem Layoutmetrikobjekt keine spezielle Funktionalität benötigt. Die Assoziation wird vom Layoutfenster selbst vollständig übernommen. Dabei ist dem untergeordneten Fenster nichts von der Beziehung zwischen Layoutfenster und -metrikobjekt bekannt.

- *GetChildLayoutMetrics* erwartet ein *TWindow* **&** und ein *TLayoutMetrics* **&** als Parameter und gibt ein *BOOL* zurück. Diese Funktion sucht nach dem durch *TWindow* **&** repräsentierten untergeordneten Fenster. Sie plaziert dann die aktuelle Layoutmetrik, die mit diesem untergeordneten Fenster verbunden (assoziiert) ist, in das *TLayoutMetrics*-Objekt, das an die Funktion übergeben wurde. Wenn *GetChildLayoutMetrics* kein untergeordnetes Fenster findet, das dem an die Funktion übergebenen Fensterobjekt gleicht, gibt sie FALSE zurück.
- *RemoveChildLayoutMetrics* erwartet ein *TWindow* **&** als Parameter und gibt ein *BOOL* zurück. Diese Funktion sucht nach dem durch *TWindow* **&** repräsentierten untergeordneten Fenster. Sie entfernt dann das untergeordnete Fenster und dessen verbundene Layoutmetrik aus der zum Layoutfenster gehörigen Liste untergeordneter Fenster. Wenn *RemoveChildLayoutMetrics* kein untergeordnetes Fenster findet, das dem an die Funktion übergebenen Fensterobjekt gleicht, gibt sie FALSE zurück.

Sie müssen eine Layoutmetrik für jedes untergeordnete Fenster eines Layoutfensters bereitstellen. Das Layoutfenster geht davon aus, daß alle seine untergeordneten Fenster ein assoziiertes Layoutmetrikobjekt besitzen. Wenn ein untergeordnetes Fenster vom Layoutfenster entfernt wird oder wenn ein untergeordnetes Fensterobjekt gelöscht wird, wird auch automatisch das assoziierte Layoutmetrikobjekt entfernt.

Rahmenfenster

Rahmenfenster (Objekte der Klasse *TFrameWindow*) sind spezielle Fenster, die Client-Fenster unterstützen. Rahmenfenster sind die Basis für MDI- und SDI-Rahmenfenster, untergeordnete MDI-Fenster und (zusammen mit *TLayoutWindow*) dekorierte Rahmenfenster.

Rahmenfenster spielen in der objektorientierten Entwicklung eine wichtige Rolle: Sie erledigen Aufgaben, die sich auf die gesamte Anwendung erstrecken, so zum Beispiel erzeugen sie Menüs und Mauspaletten. Client-Fenster innerhalb eines Rahmens können für eine einzelne Aufgabe bestimmt sein. Die Änderungen, die Sie an dem Rahmenfenster vornehmen (z.B. Hinzufügen von Mauspaletten und Statuszeilen), haben keine Auswirkungen auf das Client-Fenster.

Konstruieren von Rahmenfensterobjekten

Sie können ein Rahmenfensterobjekt über einen der beiden *TFrameWindow*-Konstruktoren konstruieren. Diese beiden Konstruktoren bieten die Möglichkeit, neue Rahmenfensterobjekte zusammen mit neuen Oberflächenelementen zu erzeugen oder auch ein neues Rahmenfensterobjekt mit einem bereits existierenden Oberflächenelement zu verbinden.

Konstruieren eines neuen Rahmenfensters

Der erste Konstruktor von *TFrameWindow* dient zur Erzeugung eines vollständig neuen Rahmenfensterobjektes:

```
TFrameWindow(TWindow *parent,
             const char far *title = 0,
             TWindow *clientWnd = 0,
             BOOL shrinkToClient = FALSE,
             TModule *module = 0);
```

Dabei gilt Folgendes:

- Der erste Parameter ist das dem Fenster übergeordnete Fensterobjekt. Übergeben Sie Null, wenn das von Ihnen erzeugte Fenster das Hauptfenster ist (welches kein übergeordnetes Fenster besitzt). Benutzen Sie andernfalls einen Zeiger auf das übergeordnete Fensterobjekt. Dieser Parameter ist der einzige, den Sie auf jeden Fall übergeben *müssen*.
- Der zweite Parameter ist der Fenstertitel. Beide Konstruktoren erzeugen das neue Fensterobjekt und setzen *Attr.Title* auf den Titel-Parameter. Dieser Titel ist der String, der in der Titelleiste des Fensters erscheint. Wenn Sie beim zweiten Parameter keine Angabe machen, wird in der Titelleiste auch kein Titel angezeigt.
- Der dritte Parameter dient zur Angabe eines Client-Fensters für das Rahmenfenster. Wenn Sie für den dritten Parameter keine Angabe ma-

chen, erhält der Konstruktor standardmäßig eine Null. Null bedeutet dabei, daß kein Client-Fenster vorliegt. Übergeben Sie andernfalls einen Zeiger auf das Client-Fensterobjekt.

- Der vierte Parameter dient zur Angabe, ob das Rahmenfenster auf die Größe des Client-Fensters schrumpfen soll. Wenn Sie für den vierten Parameter keine Angabe machen, erhält der Konstruktor als Vorgabewert FALSE. FALSE bedeutet dabei, daß sich das Rahmenfenster nicht an das Client-Fenster anpassen soll.
- Der fünfte Parameter wird an den Basisklassenkonstruktor als *TModule*-Parameter des Konstruktors übergeben. Dieser Parameter hat standardmäßig den Wert Null.

Es folgen einige Beispiele für die Verwendung dieses Konstruktors:

```
void
TMyApplication::InitMainWindow()
{
  // Der Vorgabewert sieht kein Client-Fenster vor.
  SetMainWindow (new TFrameWindow(0, "Main Window"));
}

void TMyApplication::InitMainWindow() {
  // Client-Fenster ist TMyClientWindow
  SetMainWindow (new TFrameWindow(0, "Main window with client",
                                  new TMyClientWindow, TRUE));
}
```

Konstruieren eines Rahmenfensteralias

Der zweite Konstruktor von *TFrameWindow* wird benutzt, um ein bereits existierendes Oberflächenelement mit einem neuen *TFrameWindow*-Objekt zu verbinden. Dieses Objekt bezeichnet man als den *Alias* für das existierende Fenster:

```
TFrameWindow(HWND hWnd, TModule *module);
```

Dabei gilt Folgendes:

- Der erste Parameter ist das Fenster-Handle des existierenden Oberflächenelementes. Dieses ist das Fenster, welches das *TFrameWindow*-Objekt kontrolliert.
- Der zweite Parameter wird an den Basisklassenkonstruktor als *TModule*-Parameter übergeben. Dieser Parameter hat standardmäßig den Wert Null.

Das folgende Beispiel zeigt, wie ein *TFrameWindow* für ein existierendes Oberflächenelement konstruiert und dieses Fenster dann als Hauptfenster benutzt wird:

```
void
TMyApplication::AddWindow(HWND hWnd)
```

```
{
  TFrameWindow frame(hWnd);
  TFrameWindow* tmp = SetMainWindow(frame);
  ShowWindow(GetMainWindow()->HWindow, SW_SHOW);
  tmp->ShutDownWindow();
}
```

Wenn Sie für *TFrameWindow* den zweiten Konstruktor verwenden, wird das Flag *wfAlias* gesetzt. Sie können in Erfahrung bringen, ob ein Fensterelement von seinem Fensterobjekt konstruiert wurde oder ob es aktuell ein Alias ist. Dazu rufen Sie die Funktion *IsFlagSet* mit dem *wfAlias*-Flag auf. Nehmen wir zum Beispiel an, Sie wissen nicht, ob die Funktion *AddWindow* im letzten Beispiel bereits ausgeführt worden ist. Wenn Ihr Hauptfenster *kein* Alias ist, wurde *AddWindow* nicht ausgeführt. Wenn Ihr Hauptfenster *ein* Alias ist, ist *AddWindow* ausgeführt worden:

```
void
TMyApplication::CheckAddExecute()
{
  if(GetMainWindow()->IsFlagSet(wfAlias))
   // Hauptfenster ist ein Alias; AddWindow wurde ausgeführt
  else
   // Hauptfenster ist kein Alias; AddWindow wurde nicht ausgeführt
}
```

Ändern von Rahmenfenstern

Viele Eigenschaften eines Rahmenfensters können gesetzt werden, nachdem das Objekt konstruiert worden ist. Sie können Objektattribute abfragen und verändern. Sie können aber ebenso die Funktionen von *TWindow* verwenden. *TFrameWindow* bietet eine Reihe von Funktionen für den Aufbau, die Änderung und das Inspizieren eines Fenstermenüs:

- *AssignMenu* wird typischerweise benutzt, um das Menü eines Fensters aufzubauen, bevor das Oberflächenelement erzeugt wurde. Dies ist beispielsweise in der Funktion *InitMainWindow* oder beim im Konstruktor des Fensterobjektes oder in der Funktion *SetupWindow* der Fall.
- *SetMenu* setzt das Menü-Handle des Fensters auf den übergebenen Parameter HMENU.
- *SetMenuDescr* setzt die Menübeschreibung des Fensters auf den übergebenen Parameter *TMenuDescr*.
- *GetMenuDescr* gibt die aktuelle Menübeschreibung zurück.
- *MergeMenu* vereinigt die aktuelle Menübeschreibung mit dem übergebenen Parameter *TMenuDescr*.
- *RestoreMenu* stellt das Menü eines Fensters aus *Attr.Menu* wieder her.
- *SetIcon* setzt das Symbol in dem als ersten Parameter übergebenen Modul auf das Symbol, das als eine Ressource im zweiten Parameter übergeben wurde.

Für weiterführende Informationen zu diesen Funktionen siehe auch das *ObjectWindows Referenzhandbuch*.

Dekorierte Rahmenfenster

Dieser Abschnitt beschreibt dekorierte Rahmenfenster. Dekorierte Rahmenfenster sind in der Klasse *TDecoratedFrame* gekapselt, die von *TFrameWindow* und *TLayoutWindow* abgeleitet ist. Dekorierte Rahmenfenster bieten die gesamte Funktionalität von Rahmenfenstern und Layout. Darüber hinaus stellen sie bereit:

- Unterstützung für das Hinzufügen von Dialogelementen (Dekorationen genannt) in den Fensterrahmen.
- Automatische Anpassung des Fensters, um Platz für die Dekorationen bereit zu stellen.

Konstruieren von dekorierten Rahmenfensterobjekten

TDecoratedWindow verfügt über genau einen Konstruktor. Mit Ausnahme des vierten Parameters ist dieser nahezu identisch mit dem ersten Konstuktor von *TFrameWindow.*

```
TDecoratedFrame(TWindow        *parent,
                const char far *title,
                TWindow        &clientWnd,
                BOOL            trackMenuSelection = FALSE,
                TModule         *module = 0);
```

Dabei gilt Folgendes:

- Der erste Parameter ist ein Zeiger auf das dem Fenster übergeordnete Fensterobjekt. Verwenden Sie Null, wenn Sie ein Hauptfenster erzeugen wollen (welches kein übergeordnetes Fensterobjekt besitzt). Verwenden Sie andernfalls einen Zeiger auf das übergeordnete Fensterobjekt. Dieser Parameter ist der einzige, den Sie übergeben *müssen*.
- Der zweite Parameter ist der Fenstertitel. Beide Konstruktoren erzeugen das neue Fensterobjekt und setzen *Attr.Title* auf den Titel-Parameter. Dieser Titel ist der String, der in der Titelleiste des Fensters erscheint. Wenn Sie beim zweiten Parameter nichts angeben, wird auch in der Titelleiste nichts angezeigt.
- Der dritte Parameter dient zur Angabe eines Client-Fensters für das Rahmenfenster. Wenn Sie beim dritten Parameter keine Angabe machen, erhält der Konstruktor als Vorgabewert eine Null. Die Null bedeutet dabei, daß kein Client-Fenster vorliegt. Übergeben Sie andernfalls einen Zeiger auf das Client-Fensterobjekt.

- Der vierte Parameter dient zur Angabe, ob Menükommandos mitverfolgt werden sollen. Wenn dies der Fall ist, versucht das Fenster, einen String an die Statuszeile des Fensters zu übergeben. Der übergebene String besitzt denselben Ressourcen-Namen wie die aktuell ausgewählte Menüoption. Sie sollten die Mitverfolgung von Menüauswahlen nicht anschalten, solange keine Statuszeile in Ihrem Fenster definiert wurde. Wenn Sie keine Angabe machen, erhält der Konstruktor als Vorgabewert FALSE. FALSE bedeutet dabei, daß Menükommandos nicht mitverfolgt werden sollen.
- Der fünfte Parameter wird an den Basisklassenkonstruktor als *TModule*-Parameter übergeben. Dieser Parameter hat standardmäßig den Wert Null.

Hinzufügen von Dekorationen zu dekorierten Rahmenfenstern

Um die grundlegenden Attribute eines dekorierten Rahmenfensters zu ändern, können Sie die auf den Seiten 158 f. beschriebenen Methoden zum Ändern von Fenstern verwenden. *TDecoratedFrame* bietet eine zusätzliche Möglichkeit, Dekorationen über die Elementfunktion *Insert* hinzuzufügen.

Um die Funktion *Insert* benutzen zu konnen, müssen Sie zuerst ein einzufügendes Dialogelement konstruieren. Gültige Dialogelemente umfassen Dialogelementleisten (*TControlBar*), Statuszeilen (*TStatusBar*), Schalter-Gadgets (*TButtonGadget*) und jeden anderen Dialogelementtyp, der auf *TWindow* basiert.

Nachdem das Dialogelement konstruiert worden ist, können Sie es über die Funktion *Insert* in das dekorierte Rahmenfenster einfügen. Die Funktion *Insert* erwartet zwei Parameter: eine Referenz auf das Dialogelement und eine Plazierungsangabe. *TDecoratedFrame* definiert den **enum**-Typ *TLocation* mit vier möglichen Werten: Top, Bottom, Left und Right.

Angenommen, Sie wollen eine Statuszeile konstruieren, die Sie am unteren Rand Ihres Rahmenfensters hinzufügen wollen. Der Code könnte dann wie folgt aussehen:

```
TStatusBar* sb = new TStatusBar(0, TGadget::Recessed,
                                TStatusBar::CapsLock |
                                TStatusBar::NumLock |
                                TStatusBar::Overtype);


TDecoratedFrame* frame = new TDecoratedFrame(0,
                                             "Decorated Frame",
                                             *(TWindow *) 0,
                                             TRUE);

frame->Insert(*sb, TDecoratedFrame::Bottom);
```

MDI-Fenster

MDI-Fenster sind Teil der MDI-Schnittstelle für die Verwaltung mehrerer Fenster, sowie für die Verwaltung verschiedener Darstellungen bei einer einzigen Anwendung. Ein Dokument bezeichnet im allgemeinen eine datei spezifische Aufgabe wie die Bearbeitung einer Textdatei oder das Arbeiten an einer Arbeitsblattdatei.

MDI-Anwendungen

Bestimmte Komponenten sind in jeder MDI-Anwendung vorhanden. An erster Stelle ist hier das Hauptfenster zu nennen, das *MDI-Rahmenfenster* genannt wird. Innerhalb des Client-Bereiches des Rahmenfensters befindet sich das *MDI-Client-Fenster*, das untergeordnete Rahmenfenster mit Namen *untergeordnete MDI-Fenster* enthält. Bei der Verwendung von *Dok/View*-Klassen kann die Anwendung die Ansichten (Views) in den MDI-Fenstern darstellen. Siehe Kapitel 11 für weitere Informationen zu *Dok/View*-Klassen.

Das MDI-Fenstermenü

Die Menüleiste des Rahmenfensters hat eine Menüoption, die für gewöhnlich *Window* heißt und die die untergeordneten MDI-Fenster kontrolliert. Das *Window*-Menü hat in der Regel Einträge wie *Tile*, *Cascade*, *Arrange* und *Close All*. Der Name von jedem offenen, untergeordneten MDI-Fenster wird automatisch an das Ende dieses Menüs angefügt. Das aktuell ausgewählte Fenster ist dabei markiert.

Untergeordnete MDI-Fenster

Untergeordnete MDI-Fenster besitzen einige Merkmale von überlappten Fenstern. Ein untergeordnetes MDI-Fenster kann auf die volle Größe seines MDI-Client-Fensters vergrößert werden oder es kann als Symbol dargestellt werden, das sich innerhalb des Client-Fensters befindet. Untergeordnete MDI-Fenster erscheinen niemals außerhalb ihrer Client- oder Rahmenfenster. Da untergeordnete MDI-Fenster keine zugehörigen Menüs besitzen, können sie über ein *TMenuDescr*-Objekt verfügen, welches das Rahmenfenster als Menü benutzt, sobald das untergeordnete Fenster aktiv ist. Der Titel eines jeden untergeordneten MDI-Fensters ist oft der mit dem Fenster assoziierte Dateiname. Dieses Verhalten ist jedoch nur optional und unterliegt Ihrer Kontrolle.

MDI in ObjectWindows

ObjectWindows definiert Klassen für jeden MDI-Fenstertyp:

- *TMDIFrame*
- TMDIClient
- TMDIChild

In ObjectWindows ist das MDI-Rahmenfenster der Besitzer des MDI-Client-Fensters und das MDI-Client-Fenster der Besitzer aller untergeordneten MDI-Fenster.

Die Elementfunktionen von *TMDIFrame* verwalten das Rahmenfenster und dessen Menü. ObjectWindows übergibt Kommandos zuerst an das aktive Fenster und dann an dessen übergeordnetes Fenster. Damit kann das Client-Fenster die Menükommandos des Rahmenfensters verarbeiten. Da *TMDIFrame* keine speziellen Elementfunktionen braucht, müssen Sie nur selten Ihre eigene MDI-Rahmenfensterklasse ableiten. Verwenden Sie stattdessen Instanzen von *TMDIFrame*. Da *TMDIChild* von *TFrameWindow* abgeleitet ist, kann es als Rahmenfenster mit einem Client-Fenster benutzt werden. So können Sie spezielle Fenster erzeugen, die in einem *TMDIChild*-Objekt als Client-Fenster eingesetzt werden, oder Sie können direkt spezielle *TMDIChild*-Fenster erzeugen. Vorzugsweise sollten sie spezielle Clients mit der Standardklasse *TMDIChild* verwenden. Die Wahl liegt bei Ihnen und hängt von Ihrer speziellen Anwendung ab.

Erzeugen einer MDI-Anwendung

In ObjectWindows können Sie auf einfache Weise eine MDI-Anwendung erzeugen. Gehen Sie dazu nach folgenden Schritten vor:

- Erzeugen Sie ein MDI-Rahmenfenster.
- Entwickeln Sie die Funktionalität für das MDI Client-Fenster.
- Erzeugen Sie untergeordnete MDI-Fenster.

Die *TMDIXxx*-Klassen von ObjectWindows übernehmen das MDI-spezifische Verhalten für Sie, so daß Sie sich auf das von Ihnen gewünschte anwendungsspezifische Verhalten konzentrieren können.

Erzeugen eines MDI-Rahmenfensters

Das MDI-Rahmenfenster ist immer das Hauptfenster der Anwendung. Damit sollte es in der Elementfunktion *InitMainWindow* des Anwendungsobjektes konstruiert werden. MDI-Rahmenfenster unterscheiden sich von anderen Rahmenfenstern wie folgt:

- Ein MDI-Rahmen ist immer ein Hauptfenster. Damit hat er niemals ein übergeordnetes Fenster. Der Konstruktor von *TMDIFrame* erwartet daher keinen Zeiger auf ein übergeordnetes Fensterobjekt als Parameter.
- Ein MDI-Rahmen muß ein Menü besitzen. Damit erwartet der Konstruktor von *TMDIFrame* ein Menü als Parameter. Bei nicht-MDI-Hauptrahmenfenstern sollten Sie *AssignMenu* aufrufen, um das Menü zu bestimmen. Der Konstruktor von *TMDIFrame* übernimmt diesen Aufruf für Sie. *TMDIFrame::AssignMenu* sucht in dem Menü nach dem Menü für das untergeordnete Fenster. Dazu wird nach gewissen Menükommando-IDs gesucht. Wenn die Funktion ein Window-Menü findet, werden die Titel der neuen untergeordneten Fenster automatisch an das Ende des Menüs angefügt.

Eine typische *InitMainWindow*-Funktion für eine MDI-Anwendung könnte wie folgt aussehen:

```
void TMDIApp::InitMainWindow()
{
  SetMainWindow(new TMDIFrame("MDI App", ID_MENU, *new TMyMDIClient));
}
```

Das Beispiel erzeugt ein MDI-Rahmenfenster mit dem Titel "MDI App" mit einem Menü von der ID_MENU-Ressource. Das ID_MENU-Menü sollte ein Menü für ein untergeordnetes Fenster haben. Das MDI-Client-Fenster wird von der Klasse *TMyMDIClient* erzeugt.

Erweitern des Verhaltens eines MDI-Client-Fensters

Da Sie in der Regel für Ihr MDI-Rahmenfenster eine Instanz von *TMDIFrame* verwenden, müssen Sie Ihrer MDI-Client-Fensterklasse anwendungsbezogenes Verhalten hinzufügen. Das Rahmenfenster besitzt Menüs und Mauspaletten, jedoch gibt es die von diesen generierten Kommandos an das Client-Fenster und an die Anwendung weiter. Eine gemeinsame Antwortfunktion auf die Botschaft des Menükommandos *Datei | Öffnen*, würde mit dem Öffnen eines anderen untergeordneten MDI-Fensters antworten.

Manipulieren von untergeordneten Fenstern

TMDIClient enthält mehrere Elementfunktionen für die Manipulation von untergeordneten MDI-Fenstern. Die Kommandos von dem Menu für untergeordnete Fenster in der MDI-Anwendung kontrollieren die untergeordneten Fenster. *TMDIClient* antwortet automatisch auf solche Kommandos und führt die ihnen angemessene Aktion aus:

Tabelle 8.3: Standardmenüverhalten für untergeordnete MDI-Fenster

Aktion	Menükommando-ID	TMDIClient-Elementfunktion
Cascade	CM_CASCADECHILDREN	*CmCascadeChildren*
Tile	CM_TILECHILDREN	*CmTileChildren*
Tile Horizontally	CM_TILECHILDRENHORIZ	*CmTileChildrenHoriz*
Arrange Icons	CM_ARRANGEICONS	*CmArrangeIcons*
Close all	CM_CLOSECHILDREN	*CmCloseChildren*

Die Header-Datei OWL\MDI.H schließt OWL\MDI.RH für Ihre Anwendungen mit ein. OWL\MDI.RH ist eine Ressourcen-Header-Datei, die die oben aufgeführten Menükommando-IDs definiert. Wenn Sie Ihre Menüs im Resource Workshop entwerfen, sollten Sie OWL\MDI.RH einbinden, um diese IDs verfügbar zu haben.

Untergeordnete MDI-Fenster sollten auf keines der Menükommandos für untergeordnete Fenster antworten. Das MDI-Client-Fenster übernimmt diese Aufgabe.

Erzeugen eines untergeordneten MDI-Fensters

Es gibt zwei Möglichkeiten, ein untergeordnetes MDI-Fenster zu erzeugen: automatisch in *TMDIClient::InitChild* oder manuell an einer anderen Stelle.

Automatisches Erzeugen untergeordneter Fenster

TMDIClient definiert die Antwortfunktion *CmCreateChild*, um auf die Botschaft CM_CREATECHILD zu antworten. *CmCreateChild* wird im allgemeinen benutzt, um auf das Menükommando *Datei|Neu* einer MDI-Anwendung zu antworten. *CmCreateChild* ruft *CreateChild* auf, die wiederum *InitChild* aufruft, um ein untergeordnetes MDI-Fensterobjekt zu konstruieren. Zuletzt wird die Elementfunktion *Create* des Objektes aufgerufen, um ein Oberflächenelement für das untergeordnete MDI-Fenster zu erzeugen.

Wenn Ihre Anwendung auf CM_CREATECHILD als Kommando-ID für die Erzeugung eines neuen untergeordneten MDI-Fensters zurückgreift, dann sollten Sie *InitChild* in Ihrer MDI-Client-Fensterklasse überschreiben, damit untergeordnete MDI-Fenster konstruiert werden, sobald ein Benutzer dieses Kommando wählt:

```
TMDIChild*
TMyMDIClient::InitChild()
{
   return new TMDIChild(*this, "MDI child window");
}
```

Da der Konstruktor von *TMDIChild* eine Referenz und keinen Zeiger auf sein übergeordnetes Fensterobjekt erwartet, müssen Sie den **this**-Zeiger dereferenzieren.

Manuelle Erzeugung untergeordneter Fenster

Sie müssen die untergeordneten MDI-Fensterobjekte nicht in *InitChild* konstruieren, da Sie dies an anderer Stelle erledigen können. In diesem Fall müssen Sie jedoch dessen Oberflächenelement selbst erzeugen:

```
void
TMyMDIClient::CmFileOpen()
{
   new TMDIChild(*this, "")->Create();
}
```

K A P I T E L 9

Menüobjekte

ObjectWindows Menüobjekte umfassen Menüquellen und liefern eine Schnittstelle, um das Menü zu steuern und zu verändern. Viele Anwendungen benutzen nur ein einiges dem Hauptfenster während der Initialisierung zugeschriebenes Menü. Andere Anwendungen verlangen jedoch eine kompliziertere Menüverwaltung. In den Klassen *TMenu*, *TSystemMenu*, *TPopupMenu* und *TMenuDescr* ObjectWindows Menüobjekte eröffnen Ihnen einen einfachen Weg Menüs zu erzeugen und zu manipulieren, von grundlegender Zweckmäßigkeit bis hin zu komplexer Menüzusammenführung.

Dieses Kapitel behandelt folgende Aufgaben ,die mit Menüobjekten bewältigt werden können:

- Konstruktion von Menüobjekten
- Modifizieren von Menüobjekten
- Abfrage von Menüobjekten
- Verwendung von Systemmenüobjekten
- Verwendung von Pop-Up Menüobjekten
- Verwendung von Menüobjekten mit Rahmenfenstern

Konstruktion von Menüobjekten

TMenu besitzt mehrere Konstruktoren, die Menüobjekte aus existierenden Fenstern oder aus Menüquellen zu erzeugen. Nachdem ein Menü erzeugt wurde, kann es durch die *TMenu* Elementfuntionen hinzugefügt, gelöscht oder geändert werden. Tabelle 9.1 zählt die Konstruktoren auf, die dazu dienen können, Menüobjekte zu erzeugen.

Tabelle 9.1: Menüobjekte erzeugende Konstruktoren von *TMenu*

***TMenu* Konstruktoren**	**Beschreibung**
TMenu()	Erzeugt ein leeres Menü
TMenu(HWND)	Erzeugt ein Menüobjekt, das das Menü des aktuellen Fensters darstellt
TMenu(HMENU)	Erzeugt ein Menüobjekt aus einem bereits geladenen Menü.
TMenu(LPCVOID*)	Erzeugt ein Menüobjekt aus einer gespeicherten Menüvorlage
TMenu(HINSTANCE, TResID)	Erzeugt ein Menüobjekt aus einer Ressource

Modifizieren von Menüobjekten

Nachdem ein Menüobjekt erzeugt wurde, kann es durch die TMenu Elementfunktionen modifiziert werden. Tabelle 9.2 enthält eine Liste der Elementfunktionen, die zur Modifizierung der Menüobjekte dienen können.

Tabelle 9.2: Menüobjekte modifizierende Bausteine von *TMenu*

***TMenu* Elementfunktion**	**Beschreibung**
Menüeinträge hinzufügen:	
AppendMenu(uint, uint, const char*)	Fügt einen Menüeintrag am Ende des Menüs hinzu.
AppendMenu(uint, uint, const TBitmap&)	Fügt eine Bitmap als Menüeintrag am Ende des Menüs hinzu.
InsertMenu(uint, uint, uint, const char*)	Fügt einen Menüeintrag nach dem Menüeintrag mit der angegebenen ID hinzu.
InsertMenu(uint, uint, uint, const TBitmap&)	Fügt eine Bitmap als einen Menüeintrag nach dem Menüeintrag mit der angegebenen ID hinzu.
Modifizieren von Menüeinträgen:	
ModifyMenu(uint, uint, uint, const char*)	Ändert den gegebenen Menüeintrag.
ModifyMenu(uint, uint, uint, const TBitmap&)	Ändert den gegebenen Menüeintrag in eine Bitmap.
Aktivieren und Deaktivieren von Menüeinträgen:	
EnableMenuItem(uint, uint)	Aktiviert oder Deaktiviert den gegebene Menüeintrag
Löschen und Entfernen von Menüeinträgen:	
DeleteMenu(uint, uint)	Entfernt den Menüeintrag aus dem ihn enthaltenden Menü. Löscht ihn, wenn es sich um ein Pop-Up Menü handelt.
RemoveMenu(uint, uint)	Entfernt den Menüeintrag aus dem Menü, aber nicht aus dem Speicher.
Anwählen Menüeinträge:	
CheckMenuItem(uint, uint)	Wählt den Menüeintrag an oder ab.
SetMenuItemBitmaps(uint, uint, const TBitmap*, const TBitmap*)	Spezifiziert die bei An- oder Abwahl des gegebenen Menüeintrags darzustellende Bitmap.

Darstellung von Menüeinträgen:

TrackPopupMenu(uint, int, int, int, HWND, TRect*) TrackPopupMenu (uint, TPoint&, int, HWND, TRect*)	Stellt das Menü an der gegebenen Position als Pop-Up Menü des spezifizierten Fensters dar. Nachdem das Menüobjekt modifiziert wurde, sollte die Elementfunktion DrawMenuBar des Objektes aufgerufen werden, damit die Menüleiste aktualisiert werden kann.

Abfrage von Menüobjekten

TMenu besitzt einige Elementfunktionen und Elementoperatoren, die aufgerufen werden können, um Informationen über das Menüobjekt und seine Menüs zu erhalten. Es kann nötig sein zuerst eine abfragende Elementfunktion aufzurufen, bevor eine modifizierende Elementfunktion aufgerufen werden kann. So muß z.B. *GetMenuCheckmarkDimensions* vor *SetMenuItemBitmaps* aufgrufen werden.

Tabelle 9.3 enthält einen Überblick über die Abfrage-Elementfunktionen der Menüobjekte.

Tabelle 9.3: Menüobjekte abfragende Bausteine von TMenu

TMenu Elementfunktion	Beschreibung
Abfrage des Menüobjektes als Ganzes:	
operator: *uint()*	Sendet die Kennung des Menüs als ein uint zurück.
operator: *HMENU()*	Sendet die Kennung des Menüs als ein HMENU zurück.
IsOK()	Prüft ob das Menü richtig ist (ob die Kennung gültig ist)
GetMenuItemCount()	Ermittelt die Anzahl der Menüeinträge.
GetMenuCheckMarkDimensions(TSize&)	Ermittelt die Größe der Bitmap, die zur Darstellung der Checkmarken von angewählten Menüeinträgen benutzt Bitmap wird.
Abfrage der Einträge im Menü:	
GetMenuItemID(int)	Ermittelt die ID des Menüeintrags an der angegebenen Stelle.
GetMenuState(uint, uint)	Ermittelt die Statusflags des angegebenen Menüeintrags.
GetMenuString(uint, char*, int, uint)	Holt den Text des angegebenen Menüeintrags.
GetSubMenu(int)	Ermittelt das Handle an der angegebenen Position.

Verwendung von Menüobjekten

Die Klasse *TSystemMenu* von ObjectWindows läßt es zu, das Systemmenü eines Fensters zu verändern. *TSystemMenu* wird aus *TMenu* abgeleitet und unterscheidet sich nur in seinen Konstruktor, der ein Fensterhandle und ein **bool**'sches Flag verwendet. Wenn das Flag **true** ist, dann wird das aktuelle Systemmenü gelöscht und ein ein Systemmenüprototyp generiert, der an seine Stelle gesetzt wird. Wenn das Flag **false** ist, dann stellt das Menüobjekt das aktuelle Systemmenü. Die Voreinstellung der Flagge ist **false**.

Alle von TMenu geerbten Elementfunktionen können dazu benutzt werden, das Systemmenü zu manipulieren. Das folgende Beispiel zeigt wie ein About... Menüauswahlpunkt dem Systemmenü hinzugefügt wird.

```
void
TSysMenuFrame::SetupWindow()
{
  TFrameWindow::SetupWindow();

  // Füge dem Systemmenü einen Menüeintrag About hinzu.
  TSystemMenu sysMenu(HWindow);
  sysMenu.AppendMenu(MF_SEPARATOR, 0, (LPSTR)0);
  sysMenu.AppendMenu(MF_STRING, CM_ABOUT, "&About...");
}
```

Beachten Sie bitte, das das Systemmenü in der *SetupWindow* Funktion des Fensterobjektes modifiziert wird. Das Systemmenü sollte modifiziert werden bevor das Fenster erzeugt wird. Normalerweise ist es am einfachsten, dazu die Funktion *SetupWindow* der Basisfensterklasse zu überschreiben.

Verwendung von Pop-Up Menüobjekten

TPopupMenu kann dazu benutzt werden, ein Pop-Up Menü zu erzeugen, das zu einer exisitierenden Menüstruktur hinzugefügt werden oder an jeder beliebigen Stelle des Fensters auftauchen kann. Wie *TSystemMenu*, wird auch *TPopupMenu* aus *TMenu* abgeleitet und unterscheidet sich nur in seinem Konstruktor, der ein leeres Pop-Up Menü erzeugt. Mit der Funktion *AppendMenu* können beliebige Menüeinträge hinzugefügt werden.

Sobald sie eien Pop-Up Menü erzeugt haben, können Sie *TrackPopupMenu* dazu verwenden es als ein "freischwimmendes" Menü darzustellen. *TrackPopupMenu* erzeugt ein Pop-Up Menü an einer bestimmten Stelle in ihrem Fenster. Es gibt zwei Formen dieser Funktion.

```
bool TrackPopupMenu(uint flags, int x, int y, int rsvd, HWND wnd, TRect* re
ct = 0);

bool TrackPopupMenu(uint flags, TPoint& point, int rsvd, HWND wnd, TRect* r
ect = 0);
```

wobei:

- *flags* die relative Position des Pop-Up Menüs spezifiziert. Es kann einen der folgenden Werte annehmen:
  - TPM_CENTERALIGN
  - TMP_LEFTALIGN
  - TPM_RIGHTALIGN
  - TPM_LEFTBUTTON
  - TPM_RIGHTBUTTON
- *x* und *y* geben die Bildschirmposition des Pop-Up Menüs an. In der zweiten Form von *TrackPopupMenu* geschieht durch *point* dasgleiche, da *x* und *y* im Objekt *TPoint* enthalten sind. Das Menü wird dann abhängig vom Flaginhalt relativ zu diesem Punkt erzeugt.
- *rsvd* ist ein reservierter Wert und muß auf Null gesetzt werden.
- *wnd* ist die Fensterhandle, das Botschaften bezüglich des Menüs erhält.
- *rect* definiert den Bereich, den der Benutzer anklicken kann ohne das Menü zu entlassen.

Das folgenden Beispiel zeigt eine Fensterklasse, die ein Pop-Up Menü als Antwort auf einen rechten Maustastenklick darstellt.

```
class TPopupMenuFrame : public TFraemeWindow   {
  public:
    TPopupMenuFrame(TWindow* parent, const char *name);

  protected:
    TPopupMenu PopupMenu;
    void EvRButtonDown(uint modKeys, TPoint& point);

  DECLARE_RESPONSE_TABLE(TPopupMenuFrame);
};

DEFINE_RESPONSE_TABLE1(TSysMenuFrame, TFrameWindow)
  EV_WM_RBUTTONDOWN,
END_RESPONSE_TABLE;

TPopupMenuFrame::TPopupMenuFrame(TWindow* parent, const char *name)
  : TFrameWindow(parent, name)
{
```

```
    PopupMenu.AppendMenu(MF_STRING, CM_FILENEW, "Create new file");
    PopupMenu.AppendMenu(MF_STRING, CM_FILEOPEN, "Open file");
    PopupMenu.AppendMenu(MF_STRING, CM_FILESAVE, "Save file");

PopupMenu.AppendMenu(MF_STRING, CM_FILESAVEAS, "Save file under new name");

    PopupMenu.AppendMenu(MF_STRING, CM_PENSIZE, "Change pen size");
    PopupMenu.AppendMenu(MF_STRING, CM_PENCOLOR, "Change pen color");
    PopupMenu.AppendMenu(MF_STRING, CM_ABOUT, "&About...");
    PopupMenu.AppendMenu(MF_STRING, CM_EXIT, "Exit Program");
  }

  void
  TPopupMenuFrame::EvRButtonDown(uint /* modKeys */, TPoint& point)
  {
    PopupMenu.TrackPopupMenu(TPM_LEFTBUTTON, point, 0, HWindow);
  }
```

Verwendung von Menüobjekten mit Rahmenfenstern

Rahmenfensterobjekte von ObjectWindows (aus *TFrameWindow* und *TFrameWindow* abgeleiteten Klassen) liefern eine Anzahl von Funktionen, die dazu benutzt werden können Menüs zuzuweisen, auszutauschen und zu modifizieren. Es gibt zwei Wege um Rahmenfenstermenüs zu bearbeiten:

- Das Hauptmenü des Rahmenfensters direkt zuzuteilen oder zu ändern. Dies ist die typische Art und Weise mit Menüs zu arbeiten, wenn es nur ein einziges Menü ohne Menüzusammensetzung gibt.
- Den Menüdeskriptor des Rahmenfensters zuzuweisen und aus Clients- und untergeordneten Fenstern zusammanzusetzen. Menüdeskriptoren sind Objekte, die die Menüleiste in funtionale Gruppen teilen, und ein leichtes Zusammensetzen und Entfernen von Pop-Up Menüs erlauben.

Diese Methoden der Verwendung von Menüobjekten werden in den nächsten Abschnitten beschrieben.

Rahmenfenstern Menüquellen hinzufügen

Es war in ObjectWindows 1.0 allgemein üblich eine Menüressource direkt dem *Attr.Menu* Element eines Rahmenfensters zuzuteilen; z.B.,

```
Attr.Menu = MENU_1;
```

ObjectWindows erlaubt diese Zuteilungsweise nicht mehr. Stattdessen sollte die *AssignMenu* Funktion benutzt werden. *AssignMenu* wird in der Klasse *TFrameWindow* definiert und ist in jeder aus *TFrameWindow* abgeleiteten Klasse, wie z.B. *TMDIFrame*, *TMDIChild*, *TDecoratedFrame*, und *TFloatingFrame* enthalten.

Die Funktion *AssignMenu* übernimmt *TResId* als einzigen Parameter und gibt **true** zurück, falls die Zuteilungsoperation erfolgreich war. *AssignMenu* wird als virtual deklariert, damit es in Ihren eigenen *TFrameWindow*-abgeleiteten Klassen überschrieben werden kann. Das letzte Beispiel bei Verwendung der Funktion *AssignMenu* sieht wie folgt aus:

```
AssignMenu(MENU_1);
```

Sie können außerdem das Menü ändern, nachdem das Rahmenfenster erzeugt wurde. Um das Menü des Rahmenfensters zu ändern, rufen Sie die Funktion SetMenu des Fensterobjekts auf.

```
SetMenu(MENU_2);
```

Verwendung von Menüdeskriptoren

Die Menübearbeitung -Hinzufügen von Menüs für untergeordnete Fenster, Zusammensetzung von Menüs, u.s.w - kann eine ermüdende und verwirrende Aufgabe sein. ObjectWindows vereinfacht die Menübearbeitung mit Menüdarstellungen genannten Objekten. Menüdarstellungen teilen die Menüleiste in sechs Gruppen auf, die den üblichen Arten entsprechen, Funktionen in der Menüleiste anzuordnen:

- File (Datei)
- Edit
- Container (Container)
- Object (Objekt)
- Window (Fenster)
- Help (Hilfe)

Menüs in funktionelle Gruppen zu organisieren erleichtert es, ein neues Menü in eine exisiterende Menüleiste einzufügen. Betrachten Sie beispielsweise eine MDI Anwendung wie z.B. Step 11 aus dem ObjectWindows Lernprogramm in der Anleitung *ObjectWindows Lernprogramm*. Die Rahmen und Client-Fenster liefern Menüs, die dem Benutzer erlauben allgemeine Anwendungsfunktionen wie z.B. das Öffnen von Dateien, das Bearbeiten von Fenstern, u.s.w., auszuführen. Die untergeordneten Fenster handhaben die Menübefehle für Funktionen, die speziell für bestimmte Zeichnungen zuständig sind, wie z.B. Festsetzen von Linienweite und -farbe.

Im Lernprogramm verändert sich das Menü nicht, aber die Menüeinträge, die von den untergeordneten Fenstern gehandhabt werden, werden grau ausgemalt, wenn kein untergeordnetes Fenster vorhanden ist, um den Befehl zu bearbeiten. Andererseits könnte man auch nur die von Rahmen- und Client-Fenstern gehandhabten Menüs in die Menüleiste einfügen. Wenn dann ein untergeordnetes Fenster geöffnet wird, würden die vom untergeordneten Fenster gehandhabten Menüs in die existierende Menüleiste eingefügt. Die Abbildungen unten zeigen, wie dies für den Benutzer aussieht. Abbildung 9.1 zeigt die Anwendung ohne offene unter-

geordnete Fenster. Beachten sie, daß es nur drei Pop-Up Menüs in der Menüleiste gibt.

Abbildung 9.1: Menüdarstellungs-Anwendung ohne geöffnetes untergeordnetes Fenster

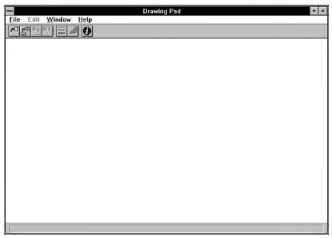
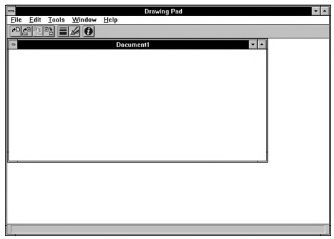


Abbildung 9.2 zeigt die Anwendung, sobald ein oder mehrere untergeordnete Fenster eröffnet wurden. Beachten Sie das neue Pop-Up Menü mit dem Namen Tools (Hilfsmittel). Das Tools-Menü wird nur in die Hauptmenüleiste eingesetzt, wenn es ein untergeordnetes Fenster gibt, in dem diese Hilfmittel verwendet werden können.

Abbildung 9.2: Menüdeskriptor-Anwendung mit geöffnetem untergeordneten Fenster



Eine Menüdarstellung einer Anwendung hinzuzufügen ist ein einfacher Vorgang.

- Setzen Sie den Menüdeskriptor für die Menüleiste des Rahmenfensters, indem Sie die Funktion *SetMenuDescr* des Rahmenfensters aufrufen.
- Wenn sie ein neues untergeordnetes Fenster erzeugen, setzen Sie die Menüdeskriptor, indem Sie die Funktion *SetMenuDescr* des untergeordneten Fensters aufrufen. Sobald das untergeordnete Fenster erzeugt wurde, setzt ObjectWindows automatisch dessen Menü des in die Rahmenfenster-Menüleiste ein, solange es aktiv ist. Beachten Sie aber, das verschiedene untergeordnete MDI-Fenster in derselben Anwendung verschiedene Menüdarstellungen haben können. Das ist sehr nützlich, wenn die untergeordneten Fenster verschiedene Arten von Dokumenten beinhalten.

Wie Menüdeskriptoren erzeugt und von Funktionen verwendet werden, wird im nächsten Abschnitt beschrieben.

Erzeugen von Menüdarstellungen

Die Klasse *TMenuDescr* implementiert die Funktionsweise von ObjectWindows Menüdeskriptoren. Menüdarstellungen nehmen eine Menüressource und unterteilen die verschieden Pop-Up Menüs in der Ressource

in sechs funktionale Gruppen. Die Benennung der Gruppen ist dahingehend beliebig, daß es keine Beschränkung dafür gibt, nur Menüs eines bestimmten funktionalen Typs in eine bestimmte Gruppe zu stecken. Die Konvention für die Namensgebung spiegelt jedoch Standardvereinbarungen für die Plazierung von Menüeinträgen wieder. Diese Namen sind in *TGroup* enum enthalten und werden in der Klasse *TMenuDescr* definiert:

- *FileGroup*
- *EditGroup*
- *ContainerGroup*
- *ObjectGroup*
- *WindowGroup*
- *HelpGroup*

Diese Gruppen werden aufeinanderfolgend von links nach rechts in der Menüleiste angeordnet. Wenn eine andere Menüdarstellung in die existierende Menüleiste eingesetzt wird, dann werden die neuen Pop-Up Menüs entsprechend Ihrer Gruppe eingefügt. Betrachten Sie z.B. die Beispiele in Abbildung 9.1 und Abbildung 9.2. Die drei ursprünglichen Pop-Up Menüs werden in die folgenden Menügruppen eingesetzt:

- Das Menü File in die Gruppe *FileGroup*
- Das Menü Window in die Gruppe *WindowGroup*
- Das Menü Help in die Gruppe *HelpGroup*

Sobald das untergeordnete Fenster erzeugt wird, wird sein Pop-Up Menü, genannt Tools, in der Gruppe *EditGroup* plaziert. Wenn die Menüs zusammengestzt werden, wird das Menü des untergeordneten Fensters automatisch zwischen das Menü File und das Menü Window eingesetzt.

Konstruktion von Menüdeskriptor-Objekten

Es gibt eine ganze Anzahl von verschiedenen *TMenuDescr* Konstruktoren. Diese werden in Tabelle 9.4 aufgezählt.

Tabelle 9.4: *TMenuDescr* Konstruktoren

Konstruktor	Funktion
TMenuDescr(TResId id, TModule module = ::Module)*	Erzeugt einen Menüdeskriptor aus der durch *id* identifizierten Menüressource. Die Gruppierung der Pop-Up Menüs wird durch Auftreten von **SEPARATOR**en auf Menüniveau (d.h. **SEPARATOR**en innerhalb eines Pop-Up Menüs werden für Gruppierungszwecke außer Acht gelassen) in der Menüressource festgelegt. Dies wird im nächsten Abschnitt detailiert behandelt.
TMenuDescr(TResId id, int fg, int eg, int cg,	Erzeugt einen Menüdeskriptor aus einer durch *id* oder *hMenu* identifzierten Menüressource. Die einzelnen Pop-Up Menüs in der

int og, int wg, int hg, *TModule* module = ::Module);*	Quelle werden dann entsprechend den Werten f*g*, *eg*, *cg*, *og*, *wg* und *hg* in den Gruppen plaziert. Die Gesamtsumme der Werte von *fg*, *eg*, *cg*, *og*, *wg* und *hg* sollte mit der Anzahl der Pop-Up Menüs in der Menüquelle übereinstimmen. Die Parameter *fg*, *eg*, *cg*, *og*, *wg* und *hg* entsprechen den in *TMenuDescr::TGroup* enum definierten Gruppen.
TMenuDescr(HMENU hMenu, *int fg, int eg, int cg,* *int og, int wg, int hg,* *TModule* module = ::Module);*	Es kann mehr als ein Pop-Up Menü in einer einzelnen Gruppe plaziert werden, und es muß nicht in jeder Gruppe eines vorkommen. Wenn Sie z.B.annehmen, daß Sie eine Menüressource mit einem Menü File, einem Menü Help und einem Menü Windows haben, die alle in der Menüressource COMMANDS enthalten sind. Sie möchten nun das Menü File in die Gruppe *FileGroup*, das Menü Window in die Gruppe *WindowGroup* und das Menü Help in die Gruppe *HelpGroup* einsetzen. Der Konstruktor würde wie folgt aussehen: `TMenuDescr md(COMMANDS, 1, 0, 0, 0, 1, 1);`
TMenuDescr()	Erzeugt einen voreingestellte Menükonstruktor ohne Menüressourcen und ohne Gruppenzugehörigkeit.
TMenuDescr(const TMenuDescr& original)	Erzeugt eine Kopie des Menüdeskriptor Objektes *original*.

Erzeugen von Menügruppen in Menüquellen TMenuDescr Klasse;Gruppierung von Pop-Up Menüs

Die Klasse *TMenuDescr* liefert zwei Wege, um Gruppen, die in die verschiedenen Pop_up Menüs gehören, festzusetzen.

- Explizites Durchnummerieren der Menüressourcen im *TMenuDescr* Konstruktor.
- Plazieren von Trennern im Pop-Up Menü-Level in der Menüressource.

Frühe Versionen von ObjectWindows lieferten nur die erste Möglichkeit. Die zweite Methode ist neu in ObjectWindows 2.5. Sie ist flexibler, da der *TMenuDescr* Konstruktor nicht mehr bei jedem Einfügen oder Entfernen eines Pop-Up Menüs in der Menüquelle modifiziert werden muß.

Um Gruppen in Ihrer Menüressource einzurichten, müssen Trenner im Pop-Up Menü-Level eingefügt werden. D.h. die Trenner werden *außerhalb* der Pop-Up Menüdefinitionen gesetzt. Diese Trenner ergeben nur für den *TMenuDescr* Konstruktor Sinn und verursachen keine Änderungen im Aussehen Ihrer Menüleiste. Trenner innerhalb der Pop-Up Menüs werden ganz normal behandelt, d.h. sie erscheinen im Pop-Up Menü als Trennlinien zwischen den Menüpunkten.

Folgendes Beispiel zeigt wie eine Menüquelle mittels Trennern in der Menüquelle in Gruppen unterteilt werden kann. Die Menüquelle wird in sechs Gruppen unterteilt. Vier dieser Gruppen enthalten eigentliche Pop-Up Menüs - die Menüs File, Edit, Window und Help. Die zwei anderen Gruppen sind leer.

```
IDM_COMMANDS MENU
{
  POPUP "File"
   {
    MENUITEM "&New\aCtrl+N", CM_FILENEW
    MENUITEM "&Open\aCtrl+O", CM_FILEOPEN
    MENUITEM "&Save\aCtrl+S", CM_FILESAVE
    MENUITEM "Save &as...", CM_FILESAVEAS
    MENUITEM SEPARATOR
    MENUITEM "&Print\aCtrl+P", CM_FILEPRINT
  }
  MENUITEM SEPARATOR
  POPUP "&Edit"
  {
    MENUITEM "&Undo\aCtrl+Z", CM_EDITUNDO
    MENUITEM Separator
    MENUITEM "&Cut\aCtrl+X", CM_EDITCUT
    MENUITEM "C&opy\aCtrl+C", CM_EDITCOPY
    MENUITEM "&Paste\aCtrl+V", CM_EDITPASTE
    MENUITEM "&Delete\aDel", CM_EDITDELETE
  }
  MENUITEM SEPARATOR
  MENUITEM SEPARATOR
  MENUITEM SEPARATOR
  POPUP "&Window"
  {
    MENUITEM "&Cascade", CM_CASCADECHILDREN
    MENUITEM "&Tile", CM_TILECHILDREN
    MENUITEM "Arrange &Icons", CM_ARRANGEICONS
    MENUITEM "C&lose All", CM_CLOSECHILDREN
    MENUITEM "Add &View", CM_VIEWCREATE
  }
  MENUITEM SEPARATOR
  POPUP "&Help"
  {
    MENUITEM "&About", CM_ABOUT
  }
}
```

Zusammensetzung von Menüs mit Menüdeskriptoren

Um Menüdeskriptoren für die Zusammensetzung von Menüs zu nutzen, muß die Menüdarstellung des Rahmenfensters vor der Erstellung des Fensters gesetzt werden. Normalerweise geschieht dies während des Aufrufs der Funktion *InitMainWindow*. Wann immer Sie das Menü eines untergeordneten Fensters in das seines Elternfensters einfügen möchten, muß der Menüdeskriptor des untergeordneten Fensters vor ihm selbst aufgebaut werden. Ist das untergeordnete Fenster einmal erzeugt, wird sein Menüdeskriptor automatisch dem des Elternfensters hinzugefügt:

Ein Menüdeskriptor eines Fensters wird mittels der Funktion *SetMenuDescr* gesetzt. *SetMenuDescr* wurde von *TFrameWindow* geerbt. Es übergibt void und nimmt eine const *TMenuDescr*-Referenz als einzigen Parameter. Das folgende Beispiel zeigt, wie Sie die Menüdeskriptoren für die Beispiele in den Abbildungen 9.1 und 9.2 setzen können.

```
class TMenuDescrApp : public TApplication
{
  public:
     TMenuDescrApp(const char* name) : TApplication(name) {}

    void InitMainWindow()
    {

SetMainWindow(Frame = new TMDIFrame(Name, COMMANDS, *new TMenuDescrMDIClien
t));
      FrameSetMenuDescr(TMenuDescr(COMMANDS));
    }

  protected:
    TMDIFrame* Frame;
};

void
TMenuDescrMDIClient::CmAddMenu1()
{

TMDIChild *child = new TMDIChild(*this, "Child Window 1", new TMenuDescrWin
dow, true);
  childSetMenuDescr(TMenuDescr(IDM_MENU1));
  childCreate();
}
```

K A P I T E L 10

Dialogfensterobjekte

Dialogfensterobjekte sind Oberflächenobjekte, die das Verhalten von Dialogfenstern kapseln. Die Klasse *TDialog* von ObjectWindows unterstützt die Initialisierung, Erzeugung und Ausführung für alle Dialogfenstertypen. Vergleichbar mit der Ableitung der Fensterobjekte von *TWindow* können Sie für jedes Dialogfenster Ihrer Anwendung spezialisierte Dialogfensterobjekte von *TDialog* ableiten.

ObjectWindows unterstützt ebenso Klassen, die die *allgemeinen Dialogfenster* von Windows kapseln. Windows stellt diese allgemeinen Dialogfenster für die Auswahl von Dateinamen, Schriften, Farben usw. zur Verfügung.

Dieses Kapitel befaßt sich mit den folgenden Themen:

- Verwenden von Dialogfensterobjekten
- Verwenden von Dialogfenstern als Hauptfenster
- Manipulieren von Dialogelementen in Dialogfenstern
- Assoziieren von Oberflächenobjekten mit Dialogelementen
- Verwenden von allgemeinen Dialogfenstern

Verwenden von Dialogfensterobjekten

Dialogfensterobjekte werden nicht in gleicher Weise wie Fensterobjekte verwendet. Dialogfenster werden einem anderen Fenster untergeordnet. Einfache Dialogfenster, die nur für eine kurze Dauer erscheinen, können Sie über eine einzige Elementfunktion des übergeordneten Fensters kontrollieren. Das Dialogfensterobjekt kann in der Elementfunktion konstruiert, ausgeführt und gelöscht werden.

Die Verwendung eines Dialogfensterobjektes erfordert die folgenden Schritte:

- Konstruieren des Objektes
- Ausführen des Dialogfensters
- Schließen des Dialogfensters

Konstruieren eines Dialogfensterobjektes

Dialogfenster werden über eine Dialogfenster-Ressource entworfen und erzeugt. Um Ihre Dialogfenster-Ressourcen zu erzeugen, können Sie den Resource Workshop von Borland verwenden. Zum Binden der Ressourcen an Ihre Anwendung steht Ihnen die Projektverwaltung der IDE, RLINK oder TLINK zur Verfügung. Die Dialogfenster-Ressource beschreibt Aussehen und Position von Dialogelementen wie Schalter, Auswahllisten, Gruppenfelder usw.. Sie hat allerdings nichts mit dem Verhalten des Dialogfensters zu tun. Dies liegt in der Verantwortung des Anwendung.

Jede Dialogfenster-Ressource besitzt einen Bezeichner. Mit Hilfe dieses Bezeichners kann das Dialogfensterobjekt die verwendete Dialogfenster-Ressource angeben. Der Bezeichner kann entweder ein String oder eine ganze Zahl sein. Sie müssen diesen Bezeichner an den Konstruktor des Dialogfensters übergeben, um festzulegen, welche Ressource das Objekt verwenden soll.

Aufruf des Konstruktors

Mit dem Aufruf des Konstruktors wird ein Dialogfensterobjekt konstruiert. Er erwartet einen Zeiger auf das übergeordnete Fensterobjekt und eine Ressource-ID (die Ressource-ID kann entweder auf einem String oder auf einer ganzen Zahl basieren):

```
TDialog dialog1(this, "DIALOG_1");
   ⋮
TDialog dialog2(this, IDD_MY_DIALOG);
```

Da Sie Dialogfensterobjekte normalerweise innerhalb der Elementfunktion eines Fensterobjektes konstruieren, ist das übergeordnete Fenster fast immer mit **this** angegeben. Wenn Sie das Dialogfensterobjekt nicht innerhalb des Fensterobjektes konstruieren, dann sollten Sie das Hauptfenster des Anwendung als das übergeordnete Fenster verwenden, da es das einzige permanent präsente Fensterobjekt einer ObjectWindows-Anwendung darstellt:

```
TDialog mySpecialDialog(GetApplication()->GetMainWindow(), IDD_DLG);
```

Eine Ausnahme hierzu liegt allerdings vor, wenn Sie ein Dialogfensterobjekt als ein Client-Fenster in einem auf *TFrameWindow* oder *TFrameWindow* basierenden Konstruktor angeben. Der Konstruktor übergibt das Dialogfensterobjekt an die Funktion *TFrameWindow::Init*, die automatisch das übergeordnete Fenster für das Dialogfenster festlegt.

Ausführen eines Dialogfensters

Die Ausführung eines Dialogfensters verhält sich analog zum Erzeugen und Anzeigen eines Fensters. Da aber Dialogfenster in der Regel für eine kürzere Zeit angezeigt werden, können einige Schritte abgekürzt werden. Dies hängt davon ab, ob das Dialogfenster ein modales oder ein nicht-modales Dialogfenster ist.

Modale Dialogfenster

Die meisten Dialogfenster sind *modal*. Währen der Anzeige eines modalen Dialogfensters kann der Benutzer dessen übergeordnetes Fenster nicht anwählen oder verwenden. Er muß das Dialogfenster vor weiteren Aktionen erst bearbeiten und schließen. Ein modales Dialogfenster friert letztlich die Ausführung der restlichen Anwendung ein. Verwenden Sie *TDialog::Execute* für die Ausführung eines modalen Dialogfensters. Wenn der Benutzer das Dialogfenster schließt, löscht die Funktion *Execute* automatisch das Objekt und gibt einen Integerwert zurück, der angibt, auf welche Weise der Benutzer das Dialogfenster geschlossen hat. Der Rückgabewert ist die ID des Dialogelementes, das der Benutzer gedrückt hat, zum Beispiel IDOK für den »OK«-Schalter oder IDCANCEL für den »Abbruch«-Schalter

Damit können Sie mit einem einzigen Aufruf das Dialogfensterobjekt erzeugen, anzeigen und löschen. Das folgende Beispiel geht davon aus, daß Sie eine Dialog-Ressource IDD_MY_DIALOG haben und daß das Dialogfenster zwei Schalter besitzt. Ein »OK«-Schalter sendet den ID-Wert IDOK und eine »Abbruch«-Schalter sendet einen anderen Wert:

```
if(TMyDialog(this, IDD_MY_DIALOG).Execute() == IDOK)
   // Benutzer hat OK gedrückt
else
   // Benutzer hat Abbruch gedrückt
```

Nach der Ausführung von *Execute* wird das *TMyDialog*-Objekt *dialog* gelöscht. Beachten Sie, daß nur das Objekt und nicht die Dialogfenster-Ressource gelöscht wird. Sie können aus einer einzigen Dialogfenster-Ressource eine beliebige Zahl von Dialogfenstern erzeugen und löschen.

Nicht-modale Dialogfenster

Im Gegensatz zu modalen Dialogfenstern können Sie bei einem geöffneten nicht-modalen Dialogfenster auch andere Fenster Ihrer Anwendung benutzen. Sie können mit nicht-modalen Dialogfenstern dem Benutzer Ihrer Anwendung ermöglichen, mit dem Ausführen von Aktionen, dem Auffinden von Information oder anderen Aktivitäten fortzufahren, während das Dialogfenster gleichzeitig genutzt wird. Verwenden Sie *TDialog::Create*, um nicht-modale Dialogfenster auszuführen. Bei *Create* müssen Sie allerdings das Dialogfenster explizit sichtbar machen, indem Sie entweder das Flag WS_VISIBLE bei den Ressourcen-Eigenschaften setzen oder das Dialogfenster über die Funktion *ShowWindow* anzeigen lassen.

Angenommen, Ihre Ressourcendatei sieht in etwa wie folgt aus:

```
DIALOG_1 DIALOG 18, 18, 142, 44
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog 1"
BEGIN
  PUSHBUTTON "Button", IDOK, 58, 23, 25, 16
END
```

Nehmen wir weiter an, daß Sie dieses nicht-modale Dialogfenster über den folgenden Code erzeugen wollen:

```
  .
  .
TDialog dialog1(this, "DIALOG_1");
dialog1.Create();
  .
  .
```

Das Dialogfenster würde nicht auf Ihrem Bildschirm erscheinen. Dazu müßten Sie eine von zwei möglichen Aktionen ausführen:

- Ändern Sie die Eigenschaft für das Dialogfenster, so daß das Flag WS_VISIBLE gesetzt ist:

  ```
  STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
  ```

- Fügen Sie die Funktion *ShowWindow* nach dem Aufruf *Create* hinzu:

  ```
    .
    .
  TDialog dialog1(this, "DIALOG_1");
  dialog1.Create();
  dialog1.ShowWindow(SW_SHOW);
    .
    .
  ```

Im Gegensatz zu *TDialog::Execute* löscht *Create* das Dialogfenster nicht automatisch bei der Rückkehr. Sie müssen explizit das Dialogfenster schließen und das Objekt löschen.Die Funktionen *TDialog::CmOk* und *TDialog::CmCancel* schließen das Dialogfenster und löschen das Objekt. Sie behandeln die Botschaften IDOK und IDCANCEL, die im allgemeinen durch die Schalter »OK« und »Abbruch« geschickt werden, in der Antworttabelle von *TDialog*. Die Funktion *CmOk* ruft *CloseWindow* auf, um das nicht-modale Dialogfenster zu schließen. Die Funktion *CmCancel* ruft *ShutDownWindow* auf. Beider Funktionen schließen das Fenster und löschen das Dialogfensterobjekt. Wenn Sie eine von beiden überschreiben, *CmOk* oder *CmCancel*, müssen Sie entweder die Funktion *CmOk* bzw. *CmCancel* aus der Basisklasse aufrufen oder selbst die Operationen zum Schließen und Löschen ausführen.

Als Alternative können Sie Ihr nicht-modales Dialogfenster im Konstruktor das zum Dialogfenster übergeordneten Fensters erzeugen. Dann dürfen Sie das Dialogfenster nur einmal erzeugen. Ferner werden alle am Zustand des Dialogfensters vorgenommene Änderungen wie seine Position, den Fokus usw. für das nächste Öffnen des Dialogfensters festgehalten.

Beachten Sie, daß dies nur bei nicht-modalen Dialogfensterobjekten funktioniert, da bei der Verwendung der Funktion *Execute* das Dialogfensterobjekt automatisch gelöscht wird. Zusätzlich müssen Sie die Funktionen *CmOk* und *CmCancel* überschreiben, damit dies geeignet funktioniert. Rufen Sie für diese Funktionen nicht die Entsprechungen aus den Basisklassen auf. Diese Funktionen löschen nämlich ebenfalls das Dialogfensterobjekt. Sie können stattdessen den Aufruf von *DestroyWindow* in Ihren Funktionen *CmOk* und *CmCancel* verwenden, um das Dialogfenster zu schließen, ohne das Dialogfensterobjekt zu löschen.

Wie jedes andere untergeordnete Fensterobjekt wird auch das Dialogfensterobjekt automatisch gelöscht, wenn sein übergeordnetes Fenster entfernt ist. Wenn Sie das für das Dialogfenster übergeordnete Fenster schließen, wird das Dialogfensterobjekt automatisch entfernt und Sie müssen dann das Objekt nicht mehr explizit löschen.

Im folgenden Fragment konstruiert der Konstruktor eines übergeordneten Fensters ein nicht-modales Dialogfenster und zeigt es an:

```
class TParentWindow : public TFrameWindow {
public:
  TParentWindow(TWindow* parent, const char* title);
  };
  .
  .
  .
void TParentWindow::CmDO_IT() {
  TDialog *dialog;
  dialog = new TDialog(this, IDD_EMPLOYEE_INFO);
  dialog->Create();
  }
```

Automatisches Erzeugen von Dialogfenstern

Sie können automatisches Erzeugen benutzen, damit ObjectWindows für Sie Ihre untergeordneten Dialogobjekte explizit erzeugt. Wenn die Objekte im Konstruktor einer von *TWindow* abgeleiteten Klasse erzeugt werden und der **this**-Zeiger als übergeordnetes Fenster angegeben wird, stellt die von *TWindow* abgeleitete Klasse eine Liste untergeordneter Fenster auf. Wenn dann die von *TWindow* abgeleitete Klasse erzeugt wird, versucht sie, alle untergeordneten Fenster in ihrer Liste zu erzeugen, sofern sie das Flag *wfAutocreate* gesetzt haben. Dadurch erscheinen die untergeordneten Fenster zur selben Zeit auf dem Bildschirm wie ihre übergeordnetes Fenster.

Über die Funktion *EnableAutoCreate* kann das Flag *wfAutocreate* gesetzt werden. Über die Funktion *DisableAutoCreate* kann es gelöscht werden.

TWindow verwendet *Create*, um seine untergeordneten Fenster automatisch zu erzeugen. Daher sind alle automatisch erzeugten Dialogfenster nicht-modale Dialogfenster.

Wie bei regulären nicht-modalen Dialogfenstern müssen Sie auch die automatisch erzeugten Dialogfenster sichtbar machen. Bei der automat-

ischen Erzeugung müssen Sie allerdings das Flag WS_VISIBLE in der Ressource-Datei setzen. Sie können nicht über die Funktion *ShowWindow* ein automatisches Erzeugen ermöglichen.

Der folgende Code zeigt, wie automatisches Erzeugen für ein Dialogfenster ermöglicht wird:

```
class TMyFrame : public TFrameWindow {
public:
   TDialog *dialog;
   TMyFrame(TWindow *, const char far *);
   };

TMyFrame::TMyFrame(TWindow *parent, const char far *title)
   Init(parent, TRUE);
   dialog = new TDialog(this, "MYDIALOG");

// Damit die nächste Zeile korrekt arbeitet, muß das Attribut
// WS_VISIBLE für die Ressource MYDIALOG angegeben werden.

   dialog->EnableAutoCreate();
   }
```

Wenn Sie diese Anwendung ausführen, wird das Dialogfenster automatisch für Sie erzeugt.

Verwaltung von Dialogfenstern

Dialogfenster unterscheiden sich von anderen untergeordneten Fenstern wie Fenster oder Dialogelemente dadurch, daß sie oft während der Lebensdauer ihrer übergeordneten Fenster viele Male angezeigt und entfernt werden und daß sie dagegen nur selten zur gleichen Zeit wie ihre übergeordneten Fenster angezeigt und entfernt werden. Im allgemeinen wird ein Dialogfenster in einer Anwendung als Antwort auf eine Menüauswahl, einen Mausklick, einer Fehlerbedingung oder ein anderes Ereignis angezeigt.

Deshalb müssen Sie sicher sein, daß keine nicht-modalen Dialogfensterobjekte immer wieder neu konstruiert werden, ohne sie zu löschen. Dies trifft auf modale Dialogfenster nicht zu, da sie mit dem Schließen automatisch gelöscht werden. Denken Sie daran, daß ein Dialogfensterobjekt beim übergeordneten Fenster in eine Liste untergeordneter Fenster eingefügt und mit dem übergeordneten Fenster auch selbst entfernt wird, sofern es innerhalb des Konstruktors des übergeordneten Fensterobjektes konstruiert worden ist.

Solange das Dialogfensterobjekt noch existiert, können Sie Daten von einem Dialogfenster jeder Zeit abfragen. Meistens erfolgt dies in der Elementfunktion *CmOK* des Dialogfensterobjektes, die aufgerufen wird, wenn der Benutzer den »OK«-Schalter des Dialogfensters drückt.

Fehlerbehandlung beim Ausführen von Dialogfenstern

Wie bei Fensterobjekten können auch die Elementfunktionen *Create* und *Execute* eines Dialogfensterobjektes zwei C++ Exceptions aufwerfen: *TXInvalidWindow* und *TXInvalidChildWindow*. *TXInvalidWindow* erscheint,

wenn das Dialogfenster nicht erzeugt werden kann. Meistens steht dann nicht ausreichend Speicherplatz zur Verfügung. *TXInvalidChildWindow* tritt auf, wenn im Dialogfenster in untergeordnete Fenster (Dialogelement) nicht erzeugt werden kann. *TXInvalidChildWindow*-Exceptions werden normalerweise dadurch verursacht, daß anwenderdefinierte Dialogelemente verwendet werden, dessen Fensterklassen nicht registriert sind.

Sie können auf den globalen Exception-Handler bauen, den ObjectWindows mit dem Start Ihrer Anwendung installiert, um *TXInvalidWindow* und *TXInvalidChildWindow* abzufangen. Sie können aber auch ebenso Ihren eigenen Exception-Handler installieren. Dazu müssen Sie einen **try** / **catch**-Block um den Code herum plazieren, den Sie schützen wollen. Wollen Sie zum Beispiel wissen, ob Ihre Funktion *DoStuff* einen Fehler hervorruft, könnte Ihr Code in etwa wie folgt aussehen:

```
try {
   DoStuff();
    }

catch(TWindow::TXInvalidWindow& e) {
    // Hier können Sie eine beliebige Exception-Behandlung vornehmen.
    MessageBox(0, e.why().c_str(),
               "Error", MB_OK);
    }
```

Die Behandlung von Exceptions ist im entsprechenden Kapitel des Borland C++ *Programmierhandbuches* beschrieben.

Schließen eines Dialogfensters

Jedes Dialogfenster muß für den Benutzer eine Möglichkeit zum Schließen vorsehen. Bei modalen Dialogfenstern ist dies meistens ein »OK«- oder »Abbruch«-Schalter oder auch beides. *TDialog* besitzt die Antwortfunktionen für Ereignisse *CmOk* und *CmCancel*, um auf diese Schalter zu antworten.

CmOk ruft *CloseWindow* auf, die wiederum *CanClose* aufruft, um festzustellen, ob das Schließen des Dialogfensters erlaubt ist. Wenn *CanClose* TRUE zurückgibt, übermittelt *CloseWindow* die Daten des Dialoges und schließt das Dialogfenster über den Aufruf von *ShutDownWindow*.

Nach der Konvention schließt der »Abbruch«-Schalter das Dialogfenster, ohne eine Aktion auszuführen. Deshalb sollte ein Test von *CanClose* nicht notwendig sein.

CmCancel ruft *ShutDownWindow* auf, die das Dialogfenster schließt. Es wird kein Test von *CanClose* durchgeführt.

Wenn Sie die Eingabe in einem Dialogfenster überprüfen wollen, können Sie die Elementfunktion *CanClose* des Dialogfensterobjektes überschreiben. Siehe auch die Beschreibung der *TInputValidator*-Klassen in Kapitel 16. Beim Überschreiben von *CanClose* sollten Sie sich vergewissern, daß Sie die Funktion *TWindow::CanClose* des übergeordneten Fensters aufrufen, da sie die *CanClose*-Aufrufe für untergeordneten Fenster übernimmt.

Verwenden von Dialogfenstern als Hauptfenster

Um ein Dialogfenster als Hauptfenster zu verwenden, sollten Sie am besten ein Rahmenfenster zum Hauptfenster machen, welches Ihr Dialogfenster als Client-Fenster besitzt. Leiten Sie dazu eine Anwendungsklasse von *TApplication* ab. Neben dem Konstruktor ist die einzige für unseren Zweck notwendige Funktion *InitMainWindow*. Konstruieren Sie in dieser Funktion ein Rahmenfensterobjekt und geben Sie das Dialogfenster als das Client-Fenster an. Im Konstruktor *TFrameWindow* mit fünf Parametern übergeben Sie als dritten Parameter einen Zeiger auf das Client-Fenster. Ihre Code könnte dann wie folgt aussehen:

```
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\dialog.h>

class TMyApp : public TApplication {
public:
   TMyApp(char *title) : TApplication(title) {}
   void InitMainWindow();
   };

void TMyApp::InitMainWindow() {
   SetMainWindow (new TFrameWindow(0, "My App",
                                   new TDialog(0, "MYDIALOG"), TRUE));
   }

OwlMain(int argc, char* argv[])
{
   return TMyApp ("My app").Run();
}
```

Der Konstruktor *TFrameWindow* schaltet für das Dialogfensterobjekt, welches Sie als Client übergeben, das automatische Erzeugen an. Dabei spielt keine Rolle, in welchem Zustand es von Ihnen übergeben wurde.

Sie müssen zudem sicher stellen, daß die Dialogfenster-Ressource bestimmte Eigenschaften besitzt:

- Ein Entfernen Ihres Dialogobjektes entfernt nicht den Rahmen. Dieser muß von Ihnen explizit entfernt werden.
- Sie können nicht weiter Ressourcen direkt dem Dialog dynamisch hinzufügen, da der Dialog nicht das Hauptfenster darstellt. Sie müssen die Ressourcen vielmehr dem Rahmenfenster hinzufügen. Nehmen wir zum Beispiel an, Sie hätten über die Funktion *SetIcon* ein Symbol Ihrem Dialog hinzugefügt. Dann müssen Sie jetzt die Funktion *SetIcon* für Ihr Rahmenfenster verwenden.

- Sie können nicht mehr weiter den Titel für Ihren Dialog einfach in der Ressource selbst angeben. Stattdessen müssen Sie den Titel über das Rahmenfenster setzen.
- Sie müssen die Eigenschaft des Dialogfensters wie folgt setzen:

  sichtbar (WS_VISIBLE)

  untergeordnetes Fenster (WS_CHILD)

  keine Schalter für Symbol- und Vollbilddarstellung, keine Zugleisten, Systemmenüs oder irgendeine der anderen Standardattribute für Rahmenfenster.

Manipulieren von Dialogelementen in Dialogfenstern

Kapitel 12 beschreibt die Verwendung von Dialogelementen im Detail. Es geht auch näher darauf ein, wie Dialogelemente in Fenstern anstatt in Dialogfenstern benutzt werden.

Fast alle Dialogfenster haben (als untergeordnete Fenster) Dialogelemente wie Editierfelder, Auswahllisten, Schalter usw.. Diese Dialogelemente werden von der Ressource des Dialogfensters erzeugt.

Zwischen einem Dialogfensterobjekt und seinen Dialogelementen existieren zwei Kommunikationsarten. In der einen Richtung muß das Dialogfenster seine Dialogelemente manipulieren, zum Beispiel für das Füllen einer Auswahlliste. In der anderen Richtung muß es auf die von den Dialogelementen generierten Botschaften antworten und verarbeiten, zum Beispiel, wenn der Benutzer einen Eintrag der Auswahlliste markiert.

Siehe Kapitel 4, um etwas über das Antworten auf Dialogelemente zu lernen.

Eine solche Kommunikation mit Dialogelementen ist nur selten notwendig. Die Dialogelementklassen von ObjectWindows stellen Elementfunktionen bereit, die dieselben Aktionen ausführen. Dieser Abschnitt beschreibt den Mechanismus zur Durchführung dieser Kommunikation, um das Verständnis dieses Prozesses zu verbessern.

Kommunikation mit Dialogelementen

Windows definiert eine Menge von Kontrollbotschaften, die von der Anwendung an Windows zurückgeschickt werden. Botschaften für Auswahllisten umfassen zum Beispiel LB_GETTEXT, LB_GETCURSEL und LB_ADDSTRING.

Kontrollbotschaften legen das spezifische Dialogelement fest und geben Informationen in *wParam*- und *lParam*-Argumenten weiter. Jedes Dialogelement in einer Dialog-Ressource besitzt eine ID, die das Dialogelement zum Empfang der Botschaft festlegen. Für das Senden einer Botschaft können Sie *SendDlgItemMessage* aufrufen. Die folgende Elementfunktion zum Beispiel fügt den angegebenen String über die Botschaft LB_ADDSTRING einer Auswahlliste hinzu:

```
void TTestDialog::FillListBox(const char far* string) {
  SendDlgItemMessage(ID_LISTBOX, LB_ADDSTRING, 0, (LPARAM)string);
  }
```

Obwohl *TListBox::AddString* im wesentlichen dasselbe wie diese Funktion durchführt und leichter zu verstehen ist, zeigt Ihnen dies die Verwendung von *SendDlgItemMessage*, um Aktionen zu erzwingen.

Assoziieren von Oberflächenobjekten mit Dialogelementen

Da ein Dialogfenster von seiner Ressource erzeugt wird, geben Sie mit dem C++ Code nicht das Aussehen oder die enthaltenen Dialogelemente an. Wenn man dadurch das Dialogfenster zwar sichtbar erzeugen kann, so wird dadurch auch die Manipulation der Dialogelement von der Anwendung aus erschwert. ObjectWindows ermöglicht die »Verbindung« oder *Assoziierung* von Dialogelementen in einem Dialogfenster mit Oberflächenobjekten. Diese Assoziierung von Dialogelementen mit Dialogelementobjekten gibt Ihnen zwei Möglichkeiten in die Hand:

- Bereitstellung spezieller Antworten auf Botschaften. Sie könnten zum Beispiel ein Editierfeld wünschen, das nur Ziffern als Eingabe erlaubt, oder einen Schalter, der nach dem Drücken Eigenschaften ändert.
- Verwendung von Elementfunktionen und Datenelementen für die Manipulation des Dialogelementes (dies ist einfacher und eher objektorientiert) anstatt die Verwendung von Botschaften für Dialogelemente (siehe »Kommunikation mit Dialogelementen« in diesem Kapitel).

Dialogelementobjekte

Um ein Dialogelementobjekt mit einem Dialogelement zu assoziieren, können sie einen Zeiger auf eine Dialogelementklassen als Datenelement definieren und ein Dialogelementobjekt im Konstruktor das Dialogfensterobjektes konstruieren. Dialogelementklassen wie *TButton* besitzen einen Konstruktor, der einen Zeiger auf das übergeordnete Fensterobjekt und die Ressource-ID des Dialogelementes erwartet. Im folgenden Beispiel assoziiert der Konstruktor von *TTestDialog* einen Schalter mit ID_BUTTON mit einem *TButton*-Objekt.

```
TTestDialog::TTestDialog(TWindow* parent, const char* resID)
  : TDialog(parent, resID),
    TWindow(parent) {
  TButton *button = new TButton(this, ID_BUTTON);
  }
```

Sie können auch Ihre eigene Dialogelementklasse definieren, die von einer existierenden Dialogelementklasse abgeleitet ist (wenn Sie spezielles Verhalten vorsehen wollen). Im folgenden Beispiel ist *TBeepButton* ein spezialisierter *TButton*, der die Standardantwort auf den Nachrichtenco-

de BN_CLICKED überschreibt. Ein *TBeepButton*-Objekt ist mit der Schalter-Ressource ID_BUTTON assoziiert.

```
class TBeepButton : public TButton {
public:
  TBeepButton(TWindow* parent, int resId) : TButton(parent, resId) {}

  void BNClicked();  // BN_CLICKED
  DECLARE_RESPONSE_TABLE(TBeepButton);
  };

DEFINE_RESPONSE_TABLE1(TBeepButton, TButton)
  EV_NOTIFY_AT_CHILD(BN_CLICKED, BNClicked),
END_RESPONSE_TABLE;

void TBeepButton::BNClicked() {
  MessageBeep(-1);
  }
  .
  .
  .
TBeepDialog::TBeepDialog(TWindow* parent, const char* name)
  : TDialog(parent, name),
    TWindow(parent) {
  button = new TBeepButton(this, ID_BUTTON);
  }
```

Beachten Sie, daß im Gegensatz zur Einrichtung eines Fensterobjektes, die zwei Schritte benötigt (Konstruktion und Erzeugung), die Assoziierung eines Oberflächenobjektes mit einem Oberflächenelement nur einen Konstruktionsschritt benötigt, da das Oberflächenelement bereits existiert: Es ist von der Ressource des Dialogfensters geladen. Sie müssen dem Konstruktor nur über die Ressource-ID mitteilen, welches Dialogelement aus der Ressource benutzt werden soll.

Einrichten von Dialogelementen

Sie können Dialogelemente nicht manipulieren, indem Sie zum Beispiel Strings in einer Auswahlliste hinzufügen oder die Schrift eines Editierfeldes einstellen, bis die Elementfunktion *SetupWindow* des Dialogfensterobjektes ausgeführt wird. Vor dem Aufruf von *TWindow::SetupWindow* in der Funktion *TDialog::SetupWindow* sind die Dialogelemente des Dialogfensters noch nicht mit den korrespondierenden Objekten assoziiert worden. Nach der Assoziierung sind für die Dialogelemente die Datenelemente *HWindow* der Objekte gültig.

Im folgenden Beispiel wird die Funktion *AddString* nicht aufgerufen, bis die Funktion *SetupWindow* der Basisklasse aufgerufen ist:

```
void TDerivedDialog::SetupWindow() {
  TDialog::SetupWindow();
  listbox->AddString("First entry");
  }
```

Verwenden von Dialogfenstern

In einer Windows-Anwendung muß der Benutzer oft nach einem Dateinamen, nach Farben oder Schriften gefragt werden. ObjectWindows enthält Klassen, die die Verwendung von Dialogfenstern, einschließlich der allgemeinen Dialogfenster von Windows, vereinfachen. Die unten stehende Tabelle listet die verschiedenen Typen von Dialogfenstern zusammen mit der jeweils einkapselnden ObjectWindows-Klasse auf.

Tabelle 10.1: Die von ObjectWindows gekapselten Dialogfenster

Typ	ObjectWindows-Klasse
Farbe	*TChooseColorDialog*
Schrift	*TChooseFontDialog*
Datei öffnen	*TFileOpenDialog*
Datei speichern	*TFileSaveDialog*
Suche String	*TFindDialog*
Eingabe vom Benutzer	*TInputDialog*
Druckabbruchdialog	*TPrinterAbortDlg*
Druckerkontrolle	*TPrintDialog*
Ersetze String	*TReplaceDialog*

Verwenden von Eingabedialogfenstern

Eingabedialogfenster sind einfache Dialogfenster, die den Benutzer zur Eingabe einer einzelnen Textzeile auffordern. Sie können Eingabedialogfenster sowohl als modale wie auch als nicht-modale Dialogfenster ausführen. In der Regel werden sie aber modal sein. Objekte zu Eingabedialogfenstern haben eine zu ihnen assoziierte Dialogfenster-Ressource, die in der Ressourcendatei OWL\INPUTDIA.RC vorliegt. Die .RC-Datei Ihrer Anwendung muß OWL\INPUTDIA.RC mit einbinden.

Wenn Sie ein Objekt zu einem Eingabedialogfenster konstruieren, geben Sie einen Zeiger auf das übergeordnete Fensterobjekt, einen Titel, eine Eingabeaufforderung, einen Textpuffer und dessen Größen an. Der Inhalt des Textpuffers ist der vorgegebene Eingabetext. Wenn der Benutzer »OK« wählt oder [*Return*] drückt, wird die eingegebene Textzeile automatisch in das Zeichen-Array übertragen. Hier ein Beispiel:

```
char patientName[33] = "";

TInputDialog (this, "Patient name",
              "Enter the patient's name:",
               patientName, sizeof(patientName)).Execute();
```

In diesem Beispiel ist *patientName* der Textpuffer, der mit der Eingabe des Benutzer gefüllt wird, wenn er »OK« wählt. Der Puffer ist für den Vorgabetext mit einem leeren String initialisiert.

Verwenden von allgemeinen Dialogfenstern

Die allgemeinen Dialogfenster kapseln die Funktionalität der allgemeinen Dialogfenster von Windows. Diese allgemeinen Dialogfenster erlauben dem Benutzer die Auswahl von Farben, Schriften und Dateinamen, das Suchen und Ersetzen von Strings, die Angabe von Druckoptionen und anderes mehr. Sie werden auf ähnliche Weise konstruiert und ausgeführt. Der Inhalt dieses Abschnittes befaßt sich mit den gemeinsamen Aufgaben. Die danach folgenden Abschnitte gehen näher auf die für jeden Typ von allgemeinem Dialogfenster spezifischen Aufgaben ein.

Konstruieren eines allgemeinen Dialogfensters

Jede Klasse eines allgemeinen Dialogfensters besitzt eine geschachtelte Klasse mit Namen *TData*. *TData* beinhaltet einige verwaltende Elemente und Daten, die für jeden Typ von allgemeinem Dialogfenster spezifisch sind. So besitzt zum Beispiel *TChooseColorDialog::TData* Elemente für die gewählte Farbe und ein Array für eine Menge von anwenderdefinierten Farben. Die folgende Tabelle führt die beiden Elemente auf, die allen verschachtelten *TData*-Klassen gemeinsam sind.

Tabelle 10.2: *TData*-Elemente für allgemeine Dialogfenster

Name	Typ	Beschreibung
Flags	DWORD	Eine Menge von Flags, die für allgemeine Dialogfenster spezifisch sind und die Erscheinen und Verhalten des Dialogfensters kontrollieren. CC_SHOWHELP ist zum Beispiel ein Flag, das dem allgemeinen Dialogfenster für Farbe mitteilt, einen »Hilfe«-Schalter für kontextbezogene Hilfe anzuzeigen, den der Benutzer drücken kann. Vollständige Informationen zu den verschiedenen Flags findet sich im ObjectWindows *Referenzhandbuch*.
Error	DWORD	Ein Fehlercode, wenn ein Fehler während der Bearbeitung des allgemeinen Dialogfensters auftrat; Null, wenn kein Fehler auftrat. *Execute* gibt IDCANCEL zurück, wenn der Benutzer »Abbruch« gewählt hat und wenn ein Fehler auftrat. Daher sollten Sie *Error* überprüfen, um festzustellen, ob ein Fehler tatsächlich vorliegt.

Jede Klasse eines allgemeinen Dialogfensters besitzt einen Konstruktor, der einen Zeiger auf das übergeordnete Fensterobjekt erwartet, eine Referenz auf die verschachtelte *TData*-Klasse dieser Klasse sowie optionale Parameter für eine anwenderdefinierte Dialogfenstervorlage, einen Titel-String und Modulzeiger.

Hier ist ein Beispielfragment, welches ein allgemeines Dialogfenster zur Farbauswahl konstruiert:

```
TChooseColorDialog::TData data;
static TColor custColors[16] =
{
  0x010101L;0x101010L;0x202020L;0x303030L,
```

```
  0x404040L;0x505050L;0x606060L;0x707070L,
  0x808080L;0x909090L;0xA0A0A0L;0xB0B0B0L,
  0xC0C0C0L;0xD0D0D0L;0xE0E0E0L;0xF0F0F0L
};

colors.CustColors = custColors;
colors.Flags = CC_RGBINIT;
colors.Color = TColor::Black;
if (TChooseColorDialog(this, colors).Execute() == IDOK)
  SetColor(colors.Color);
```

Wenn der Benutzer eine neue Farbe im Dialogfenster ausgewählt und den »OK«-Schlater gedrückt hat, wird diese Farbe in das Elemente *Color* des TData-Objekte abgelegt.

Ausführen allgemeiner Dialogfenster

Nach der Konstruktion eines allgemeinen Dialogfensterobjektes sollten Sie es ausführen (bei modalen Dialogfenstern) oder erzeugen (bei nicht-modalen Dialogfenstern). Die unten stehende Tabelle gibt für jeden Typ von allgemeinem Dialogfenster an, ob er modal oder nicht-modal sein muß.

Typ	modal oder nicht-modal	Aufruf über Funktion
Farbe	modal	*Execute*
Schrift	modal	*Execute*
Datei öffnen	modal	*Execute*
Datei speichern	modal	*Execute*
Suchen	nicht-modal	*Create*
Suchen/Ersetzen	nicht-modal	*Create*
Drucker	modal	*Execute*

Sie müssen den Rückgabewert von *Execute* überprüfen, um festzustellen, ob der Benutzer »OK« oder »Abbruch« gewählt hat oder ob ein Fehler auftrat:

```
TChooseColorDialog colorDlg(this, colors);
TChooseColorDialog::TData colors;

if (colorDlg.Execute() == IDOK)
  // OK: data.Color == die vom Benutzer gewählte Farbe
else
 if (data.Error)
    // Fehler aufgetreten!
    MessageBox("Error in color dialog box!", GetApplication()->Name,
               MB_OK | MB_ICONSTOP);
```

Verwenden allgemeiner Dialogfenster für Farbe

Das allgemeine Dialogfenster für Farbe ermöglicht Ihnen die Auswahl und Erzeugung von Farben für den Gebrauch in Ihrer Anwendung. Ein Malprogramm zum Beispiel könnte das allgemeine Dialogfenster für Farben benutzen, um die Farbe eines Farbtopfes auszuwählen.

TChooseColorDialog::TData besitzt mehrere Elemente, die Sie vor dem Konstruieren des Dialogfensterobjektes initialisieren müssen:

Tabelle 10.3: Datenelemente von *TChooseColorDialog::TData*

Element von *TData*	Typ	Beschreibung
Color	*TColor*	Die ausgewählte Farbe. Wenn Sie das Dialogfenster ausführen, gibt es die vorgegebene Farbe an. Wenn der Benutzer das Dialogfenster schließt, gibt es die von ihm ausgewählte Farbe an.
CustColors	*TColor**	Ein Zeiger auf ein Array mit sechzehn benutzerdefinierten Farben. Bei Eingabe gibt es vorgegebene Farben an, bei Ausgabe die vom Benutzer gewählten

Im folgenden Beispiel wird ein allgemeines Dialogfenster für Farben dazu benutzt, das Element *Color* des Fensterobjektes zu setzen. Dieses Element dient an anderer Stelle zum ausmalen des Fensters. Beachten Sie den Gebrauch der Elementfunktion *TWindow::Invalidate*, die das Neu-Malen des Fensters in der neuen Farbe initiiert.

Für Details über *TChooseColorDialog::TData::Flags* siehe das ObjetcWindows *Referenzhandbuch.*

```
void TCommDlgWnd::CmColor() {

  // verwenden Sie "static", um die benutzerdefinierten
  // Farben zwischen den Ausführungen des allgemeinen
  // Dialogfensters für Farbe festzuhalten.
  static TColor custColors[16];
  TChooseColorDialog::TData choose;

  choose.Flags = CC_RGBINIT;
  choose.Color = Color;
  choose.CustColors = custColors;

  if(TChooseColorDialog(this, choose).Execute() == IDOK)
    Color = choose.Color;
  Invalidate();
}
```

Verwenden allgemeiner Dialogfenster für Schrift

Über das allgemeine Dialogfenster für Schrift kann eine Schrift für Ihre Anwendung einschließlich der Schriftart, Größe, Eigenschaften usw. ausgesucht werden. Ein Textverarbeitung zum Beispiel kann dieses allgemeine Dialogfenster für Schrift verwenden, um die Schrift eines Absatzes auszuwählen.

TChooseFontDialog::TData enthält mehrere Elemente, die Sie vor dem Konstruieren des Dialogfensterobjektes initialisieren müssen:

Tabelle 10.4: Datenelemente von *TChooseFontDialog::TData*

Element von *TData*	Typ	Beschreibung
DC	*HDC*	Ein Handle auf einen Gerätekontext des Druckers, dessen Schrift Sie auswählen wollen, sofern Sie in *Flags* CF_PRINTERFONTS setzen. Anderenfalls wird es ignoriert.
LogFont	*LOGFONT*	Ein Handle auf ein *LOGFONT*, das das Erscheinen der Schrift angibt. Wenn Sie das Dialogfenster ausführen und das Flag CF_INITTOLOGFONTSTRUCT setzen, erscheint das Dialogfenster mit der angegebenen Schrift (oder der mit dem ähnlichsten Aussehen) als Vorgabewert. Wenn der Benutzer das Dialogfenster schließt, wird *LogFont* mit den vom Benutzer getroffenen Wahlen gefüllt.
PointSize	**int**	Punktgröße der ausgewählten Schrift (in zehntel eines Punktes). Bei Eingabe setzt das Element die Größe der Vorgabeschrift. Bei Ausgabe gibt es die vom Benutzer gewählte Größe zurück.
Color	*TColor*	Die Farbe der ausgewählten Schrift, wenn das Flag CF_EFFECTS gesetzt ist. Bei Eingabe setzt das Element die Farbe der Vorgabeschrift. Bei Ausgabe enthält es die vom Benutzer gewählte Farbe.
Style	**char far***	Erlaubt Ihnen die Angabe der Eigenschaft des Dialoges.
FontType	*WORD*	Ein Menge von Flags, die die Schriftart der ausgewählten Schrift beschreiben. Sie werden nur bei der Ausgabe gesetzt.
SizeMin	**int**	*SizeMin* und *SizeMax* geben die minimale und maximale
SizeMax	**int**	Punktgröße an (in zehntel eines Punktes) an, die der Benutzer auswählen kann, sofern das Flag CF_LIMITSIZE gesetzt ist.

Im folgenden Beispiel wird ein allgemeines Dialogfenster für Schrift dazu verwendet, das Element *Font* des Fensterobjektes zu setzen. Dieses Element wird an anderer Stelle dazu verwendet, Text in das Fenster zu schreiben. Beachten Sie, wie das Schriftobjekt mit Hilfe von *TFont* konstruiert wird.

```
void TCommDlgWnd::CmFont()
{
  TChooseFontDialog::TData FontData;

  FontData.DC = 0;
  FontData.Flags = CF_EFFECTS | CF_FORCEFONTEXIST | CF_SCREENFONTS;
  FontData.Color = Color;
  FontData.Style = 0;
  FontData.FontType = SCREEN_FONTTYPE;
  FontData.SizeMin = 0;
  FontData.SizeMax = 0;

  TChooseFontDialog *fontDlg = new TChooseFontDialog(this, FontData);
```

```
  if (TChooseFontDialog(this, FontData).Execute() == IDOK) {
    delete Font;
    Color = FontData.Color;
    Font = new TFont(&FontData.LogFont);
  }
  Invalidate();
}
```

Verwenden allgemeiner Dialogfenster für Datei öffnen

Das allgemeine Dialogfenster für Datei öffnen dient als konsistenter Ersatz für die vielen verschiedenen Typen von Dialogfenstern, die von Anwendungen zum Öffnen von Dateien benutzt wurden.

TOpenSaveDialog::TData enthält mehrere Elemente, die Sie vor dem Konstruieren des Dialogfensterobjektes initialisieren müssen. Dazu können Sie entweder den Elementen Werte zuweisen oder der Konstruktor von *TOpenSaveDialog::TData* verwenden. Er erwartet *Flags*, *Filter*, *CustomFilter*, *InitialDir* und *DefExt* (der meist gebräuchliche) als Parameter. Die Argumente haben als Vorgabewert Null.

Tabelle 10.5: Datenelemente von *TOpenSaveDialog ::TData*

Element von *TData*	Typ	Beschreibung
FileName	**char***	Der ausgewählte Dateiname. Bei Eingabe enthält das Element den vorgegebenen Dateinamen. Bei Ausgabe enthält es den ausgewählten Dateinamen.
Filter	**char***	Dateinamensfilter und Filtermuster. Jeder Filter und jedes Filtermuster haben die Form: `filter | filtermuster | ...` Dabei ist *filter* ein Text-String, der den Filter beschreibt, und *filtermuster* ist DOS-Dateiname mit Jokerzeichen. Sie können *filter* und *filtermuster* so oft wiederholen, wie Sie Filter benötigen. Sie müssen sie nur mit »|«-Zeichen trennen.
CustomFilter	**char***	Erlaubt Ihnen die Angabe selbstdefinierter Filter.
FilterIndex	**int**	Gibt an, welcher der in *Filter* vorgegebenen Filter standardmäßig angezeigt werden soll.
InitialDir	**char***	Das Verzeichnis, das beim Öffnen des Datei-Dialogfensters angezeigt werden soll. Verwenden Sie Null für das aktuelle Verzeichnis.
DefExt	**char***	Vorgegebene Dateinamenserweiterung, die an *FileName* angehängt ist, sofern der Benutzer eine Dateinamenserweiterung angibt. Wenn *DefExt* Null ist, wird keine Dateinamenserweiterung angehängt.

Im folgenden Beispiel fordert ein allgemeines Dialogfenster für Datei öffnen den Benutzer zur Eingabe eines Dateinamens auf. Wenn ein Fehler auftritt (*Execute* gibt IDCANCEL zurück und *Error* einen Wert ungleich Null), wird ein Meldungsfenster angezeigt.

```
void TCommDlgWnd::CmFileOpen()
{
  TFileOpenDialog::TData FilenameData
    (OFN_FILEMUSTEXIST | OFN_HIDEREADONLY | OFN_PATHMUSTEXIST,
    "All Files (*.*)|*.*|Text Files (*.txt)|*.txt|",
     0, "", "*");

  TFileOpenDialog *fileDlg = new TFileOpenDialog(this, FilenameData);

  if (TFileOpenDialog(this, FilenameData).Execute() != IDOK) {
    if (FilenameData.Errval) {
      char msg[50];
      wsprintf(msg, "GetOpenFileName returned Error #%ld", Errval);
      MessageBox(msg, "WARNING", MB_OK | MB_ICONSTOP);
    }
  }
}
```

Verwenden allgemeiner Dialogfenster für Datei sichern

Das allgemeine Dialogfenster für Datei sichern dient als konsistenter Ersatz für die vielen verschiedenen Typen von Dialogfenstern, die von Anwendungen zur Auswahl eines Dateinamens benutzt wurden.

TOpenSaveDialog::TData wird sowohl im allgemeinen Dialogfenster für Datei öffnen als auch in dem für Datei sichern benutzt.

Im folgenden Beispiel fordert ein allgemeines Dialogfenster für Datei sichern den Benutzer zur Eingabe eines Dateinamens auf, für den die Sicherung erfolgen soll. Das Vorgabeverzeichnis ist \WINDOWS und die vorgegebene Dateinamenserweiterung ist .BMP.

```
void TCanvasWindow::CmFileSaveAs()
{
  TOpenSaveDialog::TData data (
    OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    "Bitmap Files (*.BMP)|*.bmp|",
    0,
    "\\windows",
    "BMP");

  if (TFileSaveDialog(this, data).Execute() == IDOK) {
    // sichere Daten in eine Datei
    ifstream is(FileData->FileName);

    if (!is)
      MessageBox("Unable to open file", "File Error",
                 MB_OK | MB_ICONEXCLAMATION);
    else
      // führe Dateiaugabe durch
  }
}
```

Verwenden allgemeiner Dialogfenster für Suchen und Ersetzen

Das allgemeine Dialogfenster für Suchen und Ersetzen dient zum Suchen und optional auch zum Ersetzen von Text in den Daten Ihrer Anwendung. Diese Dialogfenster sind flexibel genug, daß sie für Dokumente wie auch für Datenbanken verwendet werden können. Der einfachste Weg, ein allgemeines Dialogfenster für Suchen und Ersetzen zu verwenden, geht über die Editierfeldklassen *TEditSearch oder TEditFile*. Sie implementieren ein Editierfeld, das Sie durchsuchen können und in dem Sie Text ersetzen können. Wenn Ihre Anwendung Text-basiert ist, können Sie das allgemeine Dialogfenster für Suchen und Ersetzen auch manuell benutzen.

Konstruieren und Erzeugen von allgemeinen Dialogfenstern für Suchen und Ersetzen

Da die Dialogfenster für Suchen und Ersetzen nicht-modal sind, halten Sie normalerweise einen Zeiger auf sie als Datenelement in Ihrem übergeordneten Fensterobjekt. Dies erleichtert die Kommunikation mit diesen Fenstern.

Die allgemeinen Dialogfenster für Suchen und Ersetzen sind nicht-modal. Sie sollten sie als Antwort auf ein Kommando (wie den Menüeintrag *Suchen | Finden* oder *Suchen | Ersetzen*) konstruieren und erzeugen. Dies zeigt das Dialogfenster an und ermöglicht dem Benutzer die Eingabe einer Suchinformation.

Siehe das ObjectWindows *Referenzhandbuch* für weitere Details über *Flags*.

TFindReplaceDialog::TData enthält die Standardelemente *Flags* sowie Elemente, die die Strings zum Suchen und Ersetzen aufnehmen.

Das folgende Beispiel verdeutlicht den Zeiger auf das Dialogfenster für Suchen im übergeordneten Fensterobjekt und zeigt die Antwortfunktion für das Kommandoereignis, die das Dialogfenster konstruiert und erzeugt.

```
class TDatabaseWindow : public TFrameWindow {
  .
  .
  .
  TFindReplaceDialog::TData SearchData;
  TFindReplaceDialog*       SearchDialog;
  .
  .
  .
};

void TDatabaseWindow::CmEditFind()
{
  // Wenn das Dialogfenster für Suchen nicht schon
  // konstruiert ist, konstruiere und erzeuge es jetzt
  if (!SearchDialog) {
    SearchData.Flags |= FR_DOWN; // standardmäßig nach unten suchen
    SearchDialog = new TFindDialog(this, SearchData);
    SearchDialog->Create();
  }
}
```

Verarbeiten von Botschaften zum Suchen und Ersetzen

Da die allgemeinen Dialogfenster für Suchen und Ersetzen nicht-modal sind, kommunizieren sie mit ihrem übergeordneten Fensterobjekt über eine registrierte Botschaft *FINDMSGSTRING*. Sie müssen eine Ereignisantwortfunktion schreiben, die auf *FINDMSGSTRING* antwortet. Diese Funktion erwartet zwei Parameter – ein *WPARAM* und ein *LPARAM* – und gibt ein *LRESULT* zurück. Der Parameter *LPARAM* beinhaltet einen Zeiger, den Sie an die Elementfunktion *UpdateData* des Dialogfensterobjektes übergeben müssen.

Nach dem Aufruf von *UpdateData* müssen Sie das Flag FR_DIALOGTERM testen. Der Code des allgemeinen Dialogfensters setzt dieses Flag, wenn der Benutzer das nicht-modale Dialogfenster schließt. Ihre Ereignisantwortfunktion sollte dann den Zeiger des Dialogfensterobjektes auf Null setzen, da er nicht länger gültig ist. Sie müssen das Dialogfensterobjekt erneut konstruieren und erzeugen.

Solange das Flag FR_DIALOGTERM nicht gesetzt war, können Sie die Botschaft *FINDMSGSTRING* verarbeiten, indem Sie die aktuelle Suche durchführen. Dies kann so einfach sein wie die Elementfunktion *Search* eines Editierfeldobjektes oder so kompliziert wie eine Suche durch eine Paradox- oder dBase-Tabelle.

Im folgenden Beispiel ist *EvFindMsg* eine Ereignisantwortfunktion für eine registrierte Botschaft. *EvFindMsg* ruft *UpdateData* auf und testet danach das Flag FR_DIALOGTERM. Wenn es nicht gesetzt war, ruft *EvFindMsg* eine andere Elementfunktion auf, um die Suche durchzuführen.

```
DEFINE_RESPONSE_TABLE1(TDatabaseWindow, TFrameWindow)
  .
  .
  .
  EV_REGISTERED(FINDMSGSTRING, EvFindMsg),
END_RESPONSE_TABLE;
  .
  .
  .
LRESULT TDatabaseWindow::EvFindMsg(WPARAM, LPARAM lParam)
{
  if (SearchDialog) {
    SearchDialog->UpdateData(lParam);
    // wird das Dialogfenster geschlossen?
    if (SearchData.Flags & FR_DIALOGTERM) {
      SearchDialog = 0;
      SearchCmd = 0;
    } else
      DoSearch();
  }
  return 0;
}
```

Behandeln des Kommandos »Weitersuchen«

Die allgemeinen Dialogfenster zum Suchen und Ersetzen haben einen »Weitersuchen«-Schalter, den der Benutzer anwählen kann, solange die Dialogfenster sichtbar sind. Die meisten Anwendungen unterstützen ebenso ein »Weitersuchen«-Kommando für das *Suchen*-Menü. Dadurch kann der Benutzer das nächste Vorkommen in einem Schritt finden und muß nicht erst das Dialogfenster zum Suchen öffnen und danach den »Weitersuchen«-Schalter drücken. *TFindDialog* und *TReplaceDialog* erleichtern es Ihnen, dieselbe Funktionalität anzubieten.

Das Setzen des Flags FR_FINDNEXT hat denselben Effekt wie das Drücken des »Weitersuchen«-Schalters:

```
void TDatabaseWindow::CmEditFindNext()
{
  SearchDialog->UpdateData();
  SearchData.Flags |= FR_FINDNEXT;
  DoSearch();
}
```

Verwenden allgemeiner Dialogfenster für Drucker

Es gibt zwei allgemeine Dialogfenster für Drucker: ein Dialogfenster *Druckauftrag*, das Ihnen eine Auswahl ermöglicht, was zu drucken ist, wohin es zu drucken ist, welche Druckqualität gewünscht wird, wieviel Kopien zu drucken sind usw.. Es gibt ferner ein Dialogfenster *Druckereinstellung*, das Ihnen eine Auswahl der im System installierten Drucker, der Seitenorientierung, der Papiergröße und -quelle ermöglicht.

Die Elemente von *TPrintDialog::TData* ermöglichen die Kontrolle über Erscheinen und Verhalten der allgemeinen Dialogfenster für Drucker:

Tabelle 10.6: Datenelemente von *TPrintDialog::TData*

Element von *TData*	Typ	Beschreibung
FromPage	**int**	Die erste Ausgabeseite, sofern das Flag PD_PAGENUMS gesetzt ist. Bei Eingabe gibt das Element die vorgegebene erste Seite an. Bei Ausgabe gibt es die vom Benutzer gewählte erste Seite an.
ToPage	**int**	Die letzte Ausgabeseite, sofern das Flag PD_PAGENUMS gesetzt ist. Bei Eingabe gibt das Element die vorgegebene letzte Seite an. Bei Ausgabe gibt es die vom Benutzer gewählte letzte Seite an.
MinPage	**int**	Die kleinste Zahl von Seiten, die der Benutzer auswählen kann.
MaxPage	**int**	Die größte Zahl von Seiten, die der Benutzer auswählen kann.
Copies	**int**	Die Zahl der zu druckenden Kopien. Bei Eingabe gibt das Element die vorgegebene Zahl von Kopien an. Bei Ausgabe gibt es die vom Benutzer gewählte Zahl von Kopien an.

Im folgenden Beispiel führt *CmFilePrint* ein allgemeines Dialogfenster für einen Standarddruckauftrag aus und verwendet die Information in *TPrintDialog::TData*, um zu bestimmen, was zu drucken ist. *CmFilePrintSetup* fügt ein Flag hinzu, um das Dialogfenster für die Druckereinstellung automatisch hervorzubringen.

```
void TCanvas::CmFilePrint()
{
  TPrintDialog::TData data;
  TPrintDialog *printDlg = new TPrintDialog(this, data);

  if (TPrintDialog(this, data).Execute() == IDOK)
    // Verwenden Sie TPrinter und TPrintout,
    // um die Darstellung zu drucken
}

void TCanvas::CmFilePrintSetup()
{
  static TPrintDialog::TData data;
  data.Flags |= PD_PRINTSETUP;

  if (TPrintDialog(this, data, 0).Execute() == IDOK)
    // drucken
}
```

K A P I T E L 11

Dok/View-Objekte

ObjectWindows 2.5 enthält einen neuen Weg, um Daten festzuhalten und zu manipulieren, das Dok/View-Modell.

Das Dok/View-Modell besteht aus drei Teilen:

- Dokumentobjekte, die viele verschiedene Typen von Daten enthalten können und die Methoden zum Zugriff auf die Daten bereitstellen
- View-Objekte, die eine Schnittstelle zwischen einem Dokumentobjekt und der Benutzeroberfläche formen und die kontrollieren, wie die Daten angezeigt werden und wie der Benutzer mit den Daten interagieren kann.
- Eine Anwendungs-weite Dokumentverwaltung, die Dokumentobjekte und korrespondierende View-Objekte verwaltet und koordiniert.

Wie Dokumente und Views zusammenarbeiten

Dieser Abschnitt beschreibt das grundlegende Paradigma des Dok/View-Modells. Wenn Sie schon mit den Konzepten hinter dem Dok/View-Modell vertraut sind oder weiterreichende technische Informationen möchten, können Sie auch direkt zu den Abschnitten zum Programmieren übergehen.

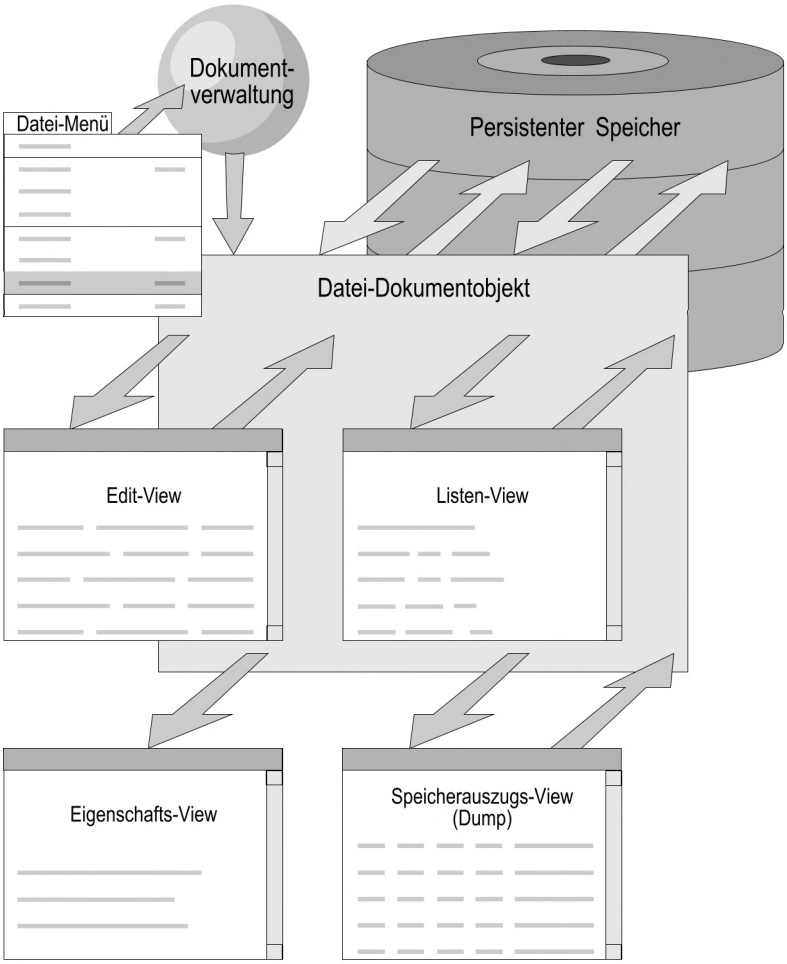
Das Dok/View-Modell befreit den Programmierer und den Benutzer von den Fragen, welcher Datentyp die Datei beinhaltet und wie diese Daten auf dem Bildschirm präsentiert werden. Beim Dok/View-Modell werden Dateitypen für Daten mit einer Dokumentklasse und einer View-Klasse assoziiert. Die Dokumentverwaltung enthält eine Liste mit Assoziationen zwischen Dokumentklassen und View-Klassen. Jede Assoziation wird *Dokument-Template* genannt (Beachten Sie, daß Dokument-Templates in *keinem* Zusammenhang mit den C++ Templates stehen.).

Eine Dokumentklasse behandelt die Speicherung und Manipulation von Daten. Sie enthält die Information, die auf dem Bildschirm angezeigt wird. Ein Dokumentobjekt kontrolliert Änderungen an den Daten sowie Zeitpunkt und Ablauf eines Datentransfers zum persistenten Speicher (wie zum Beispiel der Festplatte, einer RAM-Disk usw.).

Wenn der Benutzer ein Dokument öffnet, ob durch Erzeugen eines neuen Dokumentes oder durch Öffnen eines existierenden Dokumentes, wird das Dokument mit Hilfe einer assoziierten View-Klasse angezeigt. Die View-Klasse übernimmt die Frage, wie Daten angezeigt werden und wie der Benutzer mit den Daten auf dem Bildschirm interagiert. Letztlich bildet das View eine Schnittstelle zwischen dem angezeigten Fenster und dem Dokument. Einige Dokumente könnten nur eine assoziierte View-Klasse besitzen, einige könnten auch mehrere haben. Jeder unterschiedliche View-Typ kann dazu dienen, daß der Benutzer auf unterschiedliche Weise mit den Daten interagiert.

Die folgende Abbildung verdeutlicht die Interaktion zwischen der Dokumentverwaltung, einer Dokumentklasse und den zum Dokument assoziierten Views:

Abbildung 11.1: Dok/View-Modell-Diagramm



Diese Abbildung zeigt ein Dateidokumentobjekt von der Klasse *TFile-Document* zusammen mit einigen assoziierten Views. Die Klasse *TFile-Document* wird im Beispiel DOCVIEWX erläutert. Dieses Beispiel befindet sich im Verzeichnis *BC4*\EXAMPLES\OWL\OWLAPI\DOCVIEW, wobei *BC4* das Verzeichnis bezeichnet, in dem Sie Borland C++ 4.5 installiert haben.

Dokumentobjekte

Das traditionelle Konzept eines Dokumentes und das Dok/View-Konzept eines Dokumentes unterscheiden sich in einigen wichtigen Punkten. Das traditionelle Konzept eines Dokumentes ist generell das einer Textverarbeitungsdatei. Es besteht aus einem Text, gelegentlich gemischt mit einer Graphik und kombiniert mit eingebetteten Kommandos, die das Textverarbeitungsprogramm für die Formatierung des Dokumentes leiten.

Ein Dok/View-Dokument unterscheidet sich dagegen bedeutend vom traditionellen Dokumentkonzept:

- Die erste Unterscheidung liegt in den Inhalten der beiden Typen von Dokumenten. Wo im traditionellen Dokument meistens Text zusammen mit einigen anderen Daten vorliegt, kann ein Dok/View-Dokument buchstäblich jeden Datentyp wie Text, Grafiken, Klangdaten Multimedia-Dateien und auch andere Dokumente enthalten.
- Die nächste Unterscheidung steht im Zusammenhang mit der Präsentation. Wo das Format des traditionellen Dokumentes für gewöhnlich danach entworfen wird, wie es präsentiert werden soll, ist ein Dok/View-Dokument vollständig unabhängig davon, wie es dargestellt wird.
- Die letzt Unterscheidung besteht darin, daß ein Dokument für ein Textverarbeitungsprogramm generell von dem Format abhängt, das vom Programm gefordert wird. Die Dokumente sind dann im allgemeinen zwischen verschiedenen Textverarbeitungsprogrammen nur nach einem mühsamen Portierungsprozeß übertragbar. Ein Bestreben bei Dok/View-Dokumenten besteht in der Erleichterung der Portierung von Daten zwischen verschiedenen Anwendungen, selbst wenn deren grundsätzlicher Zweck völlig verschieden ist.

Die grundlegende Funktionalität für ein Dokument wird in der Object-Windows-Klasse *TDocument* bereitgestellt. Eine eingehendere Diskussion von *TDocument* und ihre Verwendung als eine Basis für Ihre eigenen Dokumentklassen findet sich weiter unten in diesem Kapitel.

View-Objekte

View-Objekte erlauben es den Dokumentobjekten, sich selbst nach außen hin zu präsentieren. Ohne ein View-Objekt können Sie ein Dokument weder ansehen noch manipulieren. Wenn Sie aber ein Dokument mit einem View-Objekt in einem Dokument-Template paaren, haben sie einen funk-

tionalen Teil von Daten und Code, der eine Art grafischer Repräsentation der im Dokument gespeicherten Daten und einen Weg zur Interaktion und Veränderung der Daten bereitstellt.

Die Trennung zwischen Dokument und View ermöglichen aber auch eine höhere Flexibilität bei der Frage, wann und wie die Daten im Dokument geändert werden. Während die Daten durch das View manipuliert werden, gibt das Dokument nur solche Änderungen an das Dokument zurück. Es liegt dann in der Verantwortung des Dokumentes, ob die Daten des Dokumentes geändert werden (auch *Übernehmen* der Änderungen genannt) oder ob die Änderungen verworfen werden (auch *Zurückkehren* zum Dokument genannt).

Ein anderer Vorteil der Verwendung von View-Objekten anstelle einer Art festen Darstellungsmethode (wie bei einem Textverarbeitungsprogramm) besteht darin, daß sie dem Programmierer und dem Benutzer eine Reihe unterschiedlicher Möglichkeiten bieten, dasselbe Dokument anzuzeigen und zu manipulieren. Sie könnten nur ein View für einen Dokumenttyp vorsehen, Sie könnten aber auch genauso gut drei oder vier Views.

Angenommen, Sie wollen zum Beispiel eine Dokumentklasse erzeugen, um grafische Information wie ein Bild oder eine Darstellung speichern. Für das grundlegende Produkt soll nur ein View-Typ zur Verfügung gestellt werden, der beispielsweise das Bild in einem Fenster darstellt und dann dem Benutzer das "Bemalen" und Ändern des Bildes ermöglicht. Für eine fortgeschrittene Version sollen dann extra Views existieren, die die Darstellung als Fehlfarbendarstellung, als eine hexadezimale Datei oder auch als eine Serie von Gleichungen (sofern die Darstellung mathematisch generiert wurde) präsentieren. Um auf diese anderen Views zuzugreifen, muß der Benutzer lediglich den gewünschten View-Typ beim Öffnen des Dokumentes auswählen. Das Dokument selbst ändert sich nie.

Die grundlegende Funktionalität für ein View wird in der Klasse *TView* von ObjectWindows bereitgestellt. Eine eingehendere Diskussion von *TView* und ihre Verwendung als eine Basis für Ihre eigenen View-Klassen findet sich weiter unten in diesem Kapitel.

Dokument-Templates

Eine Dokumentklasse ist mit ihrer View-Klasse (oder Klassen) über ein Dokument-Template assoziiert. Dokument-Templates sind auf einfache Weise verwendbar. Sie werden in zwei Schritten erzeugt:

Definieren Sie eine Template-Klasse, indem Sie eine Dokumentklasse mit einer View-Klasse assoziieren.

- Instanziieren Sie ein Template von einer definierten Klasse.

Der Unterschied zwischen diesen beiden Schritten ist wichtig. Wenn Sie eine Template-Klasse einmal definiert haben, können Sie eine beliebige Zahl von Instanzen dieser Template-Klasse erzeugen. Jedes Template assoziiert *nur* eine Dokumentklasse und eine View-Klasse. Jede Instanz besitzt einen Namen, eine Standarddateinamenserweiterung, ein Verzeichnis, Flags und Dateifilter. Damit können Sie eine einzelne Template-Klasse bereitstellen, die ein Dokument mit einem View assoziiert. Ferner können Sie darauf aufbauend eine Reihe unterschiedlicher *Instanzen* einer Template-Klasse erzeugen, von denen jede Dateien in verschiedenen Standardverzeichnissen oder mit verschiedenen Dateinamenserweiterungen usw. behandelt und die alle dieselben Dokument- und View-Klassen benutzen.

Dokumentverwaltung

Die Dokumentverwaltung führt die Liste von Template-Instanzen, die in Ihrer Anwendung benutzt werden, sowie eine Liste der aktuellen Dokumente. Jede Anwendung, die auf Dok/View-Dokumente zurückgreift, muß eine Dokumentverwaltung haben. Jede Anwendung kann aber zur gleichen Zeigt nur eine Dokumentverwaltung besitzen. Die Dokumentverwaltung wird einem Element der Klasse *TApplication* mit Namen *DocManager*, einem *TDocManager* *****, zugewiesen.

Die Dokumentverwaltung verbindet die Bestandteile des Dok/View-Modells: Dokumentklassen, View-Klassen und Templates. Die Dokumentverwaltung stellt ein Standard-*Datei*-Menü und eine Standardbehandlung für jede Wahl im *Datei*-Menü bereit:

Tabelle 11.1: Das *Datei*-Menü der Dokumentverwaltung

Menüauswahl	Behandlung
Neu	Erzeugt ein neues Dokument.
Öffnen...	Öffnet ein existierendes Dokument.
Speichern	Speichert das aktuelle Dokument.
Speichern unter...	Speichert das aktuelle Dokument mit einem neuen Namen.
Zurück zur Sicherung	Verwirft die Änderungen und kehrt zur zuletzt gesicherten Version des Dokumentes zurück.
Schließen	Schließt das aktuelle Dokument.
Beenden	Verläßt die Anwendung, fragt den Benutzer nach der Speicherung der Dokumente.

Haben Sie einmal Ihre Dokument- und View-Klassen geschrieben, notwendige Templates definiert und Instanzen der erforderlichen Templates erzeugt, müssen Sie nur noch Ihre Dokumentverwaltung erzeugen. Bei dieser Erzeugung richtet die Dokumentverwaltung ihre Liste von Template-Instanzen (sofern im Konstruktor angegeben) und ihr Menü ein. Danach führt sie bei jedem empfangenen Ereignis, für das sie zuständig ist, die entsprechende notwendige Aktion aus. Das Beispiel zeigt, wie eine Dokumentverwaltung für eine Anwendung eingerichtet wird.

Dokument-Templates

Dokument-Templates verbinden Dokumentklassen und View-Klassen, indem sie eine neue Klasse erzeugen. Die Dokumentverwaltung führt eine List mit Dokument-Templates, die sie für die Erzeugung einer neuen Dok/View-Instanz benutzt. Dieser Abschnitt erläutert die Erzeugung und Verwendung von Dokument-Templates.

Entwurf von Dokument-Template-Klassen

Sie können eine Dokument-Template-Klasse über das Makro DEFINE_DOC_TEMPLATE_CLASS erzeugen. Dieses Makro erwartet drei Argumente:

- Dokumentklasse
- View-Klasse
- Template-Klassenname

Die Dokumentklasse sollte diejenige sein, die Sie für die Aufnahme der Daten verwenden wollen. Die View-Klasse sollte diejenige sein, die Sie für die Darstellung der in der Dokumentklasse enthaltenen Daten verwenden wollen. Der Name der Template-Klasse sollte die Funktion des Template beschreiben. Er kann allerdings nicht ein C+++ Schlüsselwort (wie zum Beispiel **int**, **switch** usw.) oder der Name eines anderen Typs in der Anwendung sein.

Nehmen wir zum Beispiel an, Sie haben zwei Dokumentklassen – eine mit Namen *TPlotDocument*, die Grafikdaten enthält, und eine andere mit Namen *TDataDocument*, die numerische Daten enthält. Nehmen wie jetzt weiterhin an, Sie haben vier View-Klassen, zwei für jede Dokumentklasse. Für *TPlotDocument* haben Sie *TPlotView*, die die Daten in einem *TPlotDocument*-Objekt als grafische Darstellung anzeigt, und Sie haben *THexView*, die die Daten in einem *TPlotDocument*-Objekt als Arrays mit hexadezimalen Zahlen anzeigt. Für *TDataDocument* haben Sie *TSpreadView*, die die Daten in einem *TDataDocument*-Objekt wie bei einem Arbeitsblatt anzeigt, und Sie haben *TCalcView*, die die Daten in einem *TDataDocument*-Objekt nach einer Reihe von Berechnungen auf den Daten anzeigt.

Um die Dokumentklassen mit ihren Views zu assoziieren, würden Sie das Makro DEFINE_DOC_TEMPLATE_CLASS verwenden. Der Code könnte dann wie folgt aussehen:

```
DEFINE_DOC_TEMPLATE_CLASS(TPlotDocument, TPlotView, TPlotTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TPlotDocument, THexView, THexTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TDataDocument, TSpreadView, TSpreadTemplate);
DEFINE_DOC_TEMPLATE_CLASS(TDataDocument, TCalcView, TCalcTemplate);
```

Wie Sie bei der ersten Zeile sehen können, werden die existierende Dokumentklasse *TPlotDocument* und die existierende View-Klasse *TPlotView* in

einer neuen Klasse mit Namen *TPlotTemplate* miteinander verbunden und assoziiert. Dasselbe passiert in allen anderen Zeilen. Damit haben Sie vier neue Klassen: *TPlotTemplate*, *THexTemplate*, *TSpreadTemplate* und *TCalcTemplate*. Der nächste Abschnitt zeigt Ihnen, wie Sie die von Ihnen erzeugten neuen Klassen benutzen können.

Erzeugen von Instanzen einer Template-Klasse

Wenn Sie einmal eine Template-Klasse definiert haben, können Sie eine beliebige Zahl von Instanzen dieser Klasse erzeugen. Sie können Instanzen einer Template-Klasse verwenden, um verschiedene Beschreibungen eines Template bereitzustellen, nach verschiedenen Standarddateinamen zu suchen, in verschiedene Standardverzeichnisse zu schauen usw.. Sie können alle diese Dinge bewirken, wenn Sie den Konstruktor der Template-Klasse aufrufen.

Der Konstruktoraufruf einer Template-Klasse hat immer gleiches Aussehen:

```
TplName name(LPCSTR desc, LPCSTR filt, LPCSTR dir, LPCSTR ext, long flags)
```

Dabei ist:

- *TplName* der Klassenname, den Sie bei der Definition der Template-Klasse angegeben haben.
- name ein beliebiger Name, den Sie der Instanz geben.
- desc eine Textbeschreibung des Template
- filt ein String für die Filterung von Dateinamen im aktuellen Verzeichnis. Dies kann irgendein gültiger regulärer DOS-Ausdruck sein.
- dir das Standardverzeichnis für die Suche nach Dokumentdateien.
- ext die Standard-Dateinamenserweiterung für das Sichern von Dateien. Die Übergabe von Null bedeutet keine Standarderweiterung.
- flags der Modus, unter dem das Dokument geöffnet oder erzeugt werden soll. Der Modus kann eine oder mehrere der folgenden Einstellungen annehmen:

Flag	Funktion
dtAutoDelete	Das Dokumentobjekt wird gelöscht, wenn das letzte View geschlossen ist.
dtNoAutoView	Ein Standard-View wird nicht automatisch erzeugt.
dtSingleView	Es ist nur ein View je Dokument erlaubt.
dtAutoOpen	Das Dokument wird bei der Erzeugung geöffnet.
dtConfirm	Der Benutzer hat die Möglichkeit, den Pfad zu bestätigen
dtHidden	Das Template wird vor der Auswahlliste für der Benutzer versteckt.

Nehmen wir zum Beispiel an, Sie haben die folgende Definition einer Template-Klasse:

```
DEFINE_DOC_TEMPLATE_CLASS(TPlotDocument, TPlotView, TPlotTemplate);
```

Nehmen wir weiter an, Sie wollen drei Instanzen dieser Template-Klasse erzeugen:

- Eine Instanz sollte die Beschreibung "Approved plots" für Dokumentdateien mit der Erweiterung .PLT haben, die sich im Verzeichnis C:\APPROVED befinden. Sie wollen nur ein einzelnes View für das Dokument zulassen und das Dokument automatisch löschen, wenn das View geschlossen wird.
- Eine andere Instanz sollte die Beschreibung "In progress" für Dokumentdateien mit der Erweiterung .PLT haben, die sich im Verzeichnis C:\WORK befinden. Sie wollen das Dokument automatisch löschen, wenn das letzte View geschlossen wird.
- Eine andere Instanz sollte die Beschreibung "Proposals" für Dokumentdateien mit der Erweiterung .PLT oder .TMP (aber mit der Standarderweiterung .PLT) haben, die sich im Verzeichnis C:\TMP befinden. Sie wollen dieses Template versteckt halten, bis der Benutzer ein Passwort eingegeben hat, und das Dokumentobjekt löschen, wenn das letzte View geschlossen wird.

Der Code für diese Instanzen könnte dann wie folgt aussehen:

```
TPlotTemplate atpl("Approved plots",
                   "*.PLT",
                   "C:\APPROVED",
                   "PLT",
                   dtSingleView | dtAutoDelete);

TPlotTemplate btpl("In progress",
                   "*.PLT",
                   "C:\WORK",
                   "PLT",

                   dtAutoDelete);

TPlotTemplate *ctpl = new("Proposals",
                          "*.PLT; *.TMP",
                          "C:\TMP",
                          "PLT",
                          dtHidden | dtAutoDelete);
```

Beachten Sie, daß Sie wie bei jeder anderen Klasse sowohl statische als auch dynamische Instanzen eines Dokument-Templates erzeugen können.

Ändern existierender Templates

Wenn Sie einmal eine Instanz einer Template-Klasse erzeugt haben, sollte es im allgemeinen nicht mehr nötig sein, dieses Template-Objekt zu ändern. Gelegentlich könnte jedoch eine Änderung der Eigenschaften, mit

der das Template konstruiert wurde, sinnvoll sein. Dies kann über die folgenden Zugriffsfunktionen erfolgen:

- Verwenden Sie die Funktionen *GetFileFilter* und *SetFileFilter*, um den String zu ermitteln und zu setzen, mit dessen Hilfe Dateinamen im aktuellen Verzeichnis gefiltert werden.
- Verwenden Sie die Funktionen *GetDescription* und *SetDescription*, um die Textbeschreibung der Template-Klasse zu ermitteln und zu setzten.
- Verwenden Sie die Funktionen *GetDirectory* und *SetDirectory*, um das Standardverzeichnis zu ermitteln und zu setzen.
- Verwenden Sie die Funktionen *GetDefaultExt* und *SetDefaultExt*, um die Standard-Dateinamenserweiterung zu ermitteln und zu setzen.
- Verwenden Sie die Funktionen *GetFlags*, *IsFlagSet*, *SetFlag* und *ClearFlag*, um die Flag-Einstellungen zu ermitteln und zu setzen.

Verwenden der Dokumentverwaltung

Die Dokumentverwaltung, eine Instanz von *TDocManager* oder eine Instanz einer von *TDocManager* abgeleiteten Klasse, übernimmt eine Reihe von Aufgaben:

- Sie verwaltet die Liste der aktuellen Dokumente und der registrierten Templates.
- Sie bearbeitet die Standard-Kommandobotschaften des *Datei*-Menüs CM_FILENEW, CM_FILEOPEN, CM_FILESAVE, CM_FILESAVEAS, CM_FILECLOSE und optional CM_FILEREVERT.
- Sie stellt eine Schnittstelle für die Dateiauswahl bereit.

Um das Dok/View-Modell zu unterstützen, muß eine Dokumentverwaltung mit der Anwendung verbunden werden. Dies kann erfolgen, indem eine Instanz von *TDocManager* erzeugt wird und diese Instanz zur Dokumentverwaltung Ihrer Anwendungsklasse gemacht wird. Der folgende Code zeigt ein Beispiel dafür, wie eine Dokumentverwaltung mit Ihrer Anwendung verbunden wird:

```
class TMyApp : public TApplication {
public:
  TMyApp() : TApplication() {}
  void InitMainWindow() {
     .
     .
     .
     SetDocManager(new TDocManager(dmMDI | dmMenu));
     .
     .
     .
     }
  };
```

Sie können mit der Funktion *SetDocManager* die Dokumentverwaltung auf ein neues Objekt setzen. *SetDocManager* erwartet ein *TDocManager* **&** und gibt **void** zurück.

Auf die öffentlichen Daten und Funktionen der Dokumentverwaltung kann über die Funktion *GetDocManager* des Dokumentes zugegriffen werden. *GetDocManager* erwartet keine Parameter und gibt ein *TDocManager* **&** zurück. Die Dokumentverwaltung bietet die folgenden Funktionen für die Erzeugung von Dokumenten und Views:

- *CreateAnyDoc* präsentiert alle sichtbaren Templates, wogegen die Elementfunktion *CreateDoc* der Klasse *TDocTemplate* nur das eigene Template präsentiert.
- CreateAnyView filtert eine Template-Liste bis auf die Templates, die das aktuelle Dokument unterstützen, und gibt eine Liste der View-Namen aus. Dahingegen konstruiert die Elementfunktion *CreateView* der Klasse *TDocTemplate* das View, das durch die Template-Klasse der Dokumentes angegeben ist, direkt.

Spezialisierte Dokumentverwaltungen können anderen Zwecken dienen. Zum Beispiel muß ein Server der OLE 2.5 Klassenfabriken unterstützen, die Dokumente und Views über Schnittstellen erzeugen, die nicht zu ihnen gehören. Wenn der Server mit den *Embedded*-Kommandozeilen-Flags aufgerufen wird, startet er nicht seine eigene Benutzeroberfläche und kann eine Dokumentverwaltung anbinden, die die Schnittstelle durch die passende OLE-Unterstützung ersetzt.

Konstruieren einer Dokumentverwaltung

Der Konstruktor von *TDocManager* erwarteten einen einzigen Parameter. Über diesen Parameter wird der Modus der Dokumentverwaltung gesetzt. Sie können die Dokumentverwaltung in einem von zwei Modi öffnen:

- Im SDI-Modus (SDI = Single-Document Interface, Einzeldokumentschnittstelle) können Sie nur ein einziges Dokument zu einer Zeit geöffnet haben. Wenn Sie ein neues Dokument, während ein anderes schon geöffnet ist, schließt die Dokumentverwaltung das erste Dokument und ersetzt es durch das neue Dokument.
- Im MDI-Modus (MDI = Multiple-Document Interface, Multidokumentschnittstelle) können Sie eine Reihe von Dokumenten zur gleichen Zeit geöffnet haben. Jedes Dokument ist in seinem eigenen Client-Fenster enthalte. Ferner kann jedes Dokument ein einzelner Dokumenttyp sein, der durch dieselbe View-Klasse präsentiert wird, oder es kann ein einzelnes Dokument sein, das mit verschiedenen Views präsentiert wird. Es können aber auch völlig verschiedene Dokumenttypen sein.

Um die Dokumentverwaltung im SDI-Modus zu öffnen, rufen Sie den Konstruktor mit dem Parameter *dmSDI* auf. Um die Dokumentverwal-

tung im MDI-Modus zu öffnen, rufen Sie den Konstruktor mit dem Parameter *dmMDI* auf.

Es existieren noch drei weitere Parameter, die Sie ebenfalls angeben können:

- Der Parameter *dmMenu* legt fest, daß die Dokumentverwaltung ihr eigenes *Datei*-Menü installieren sollte. Damit werden ein Standard-*Datei*-Menü der Dokumentverwaltung und dessen korrespondierende Kommandos bereitgestellt.
- Der Parameter *dmSaveEnable* aktiviert das *Speichern*-Kommando im *Datei*-Menü, selbst wenn das Dokument nicht verändert wurde.
- Der Parameter *dmNoRevert* deaktiviert das *Zurückkehren*-Kommando im *Datei*-Menü.

Wenn Sie einmal die Dokumentverwaltung konstruiert haben, können Sie den Modus nicht mehr ändern. Das folgende Beispiel verdeutlicht das Öffnen der Dokumentverwaltung entweder im SDI- oder im MDI-Modus. Es verwendet Kommandozeilen-Argumente, damit der Benutzer angeben kann, ob die Dokumentverwaltung im SDI- oder im MDI-Modus geöffnet werden soll. Diese Kommandozeilen-Argumente werden im entsprechenden Kapitel im *Programmierhandbuch* erklärt.

```
class TMyApp : public TApplication {
public:
  TMyApp() : TApplication() {}
  void InitMainWindow();
  int DocMode;
  };

void TMyApp::InitMainWindow() {
  switch ((_argc > 1 && _argv[1][0]=='-' ? _argv[1][1]
                                         : (char)0) | ('S'^'s')) {
     case 's':  DocMode = dmSDI; break;    // Kommandozeile: -s
     case 'm':  DocMode = dmMDI; break;    // Kommandozeile: -m
     default :  DocMode = dmMDI; break;    // keine Kommandozeile
     }
  SetDocManager(new TDocManager(DocMode | dmMenu));
  };
```

Wenn der Benutzer damit die Anwendung mit der Option **-s** startet, öffnet die Dokumentverwaltung im SDI-Modus. Wenn der Benutzer die Anwendung mit der Option **-m** oder mit keiner Option startet, öffnet die Dokumentverwaltung im MDI-Modus.

Ereignisbehandlung von *TDocManager*

Wenn Sie den Parameter *dmMenu* bei der Konstruktion Ihres *TDocManager*-Objektes angeben, behandelt die Dokumentverwaltung bestimmte Ereignisse wegen der Dokumente mit Hilfe einer Antworttabelle für die Verarbeitung der Standardmenükommandos. Diese Menükommandos

werden von der Dokumentverwaltung bereitgestellt, auch wenn keine Dokumente geöffnet sind und ungeachtet dessen, ob Sie die Ressourcen zu Ihrer Anwendung explizit hinzufügen. Das *Datei*-Menü wird ebenso durch die Dokumentverwaltung bereitgestellt.

Die von der Dokumentverwaltung behandelten Ereignisse sind:

- CM_FILECLOSE
- CM_FILENEW
- CM_FILEOPEN
- CM_FILEREVERT
- CM_FILESAVE
- CM_FILESAVEAS
- CM_VIEWCREATE

Es kann sein, daß Sie diese Botschaften in einigen Instanzen selbst behandeln wollen. Die die Ereignistabelle der Dokumentverwaltung zuletzt durchsucht wird, können Sie diese Events auf der Ebene des View, des Rahmens oder der Anwendung behandeln. Eine andere Möglichkeit besteht darin, die Dokumentverwaltung ohne den Parameter *dmMenu* zu konstruieren. Sie müssen dann Funktionen für die Behandlung dieser Ereignisse bereitstellen. Dies erfolgt im allgemeinen durch das Anwendungsobjekt oder Ihr Oberflächenobjekt.

Sie können immer noch die Funktionen der Dokumentverwaltung über das Element *DocManager* des Anwendungsobjektes aufrufen. Nehmen wir zum Beispiel an, Sie wollen vor dem Öffnen einer Datei eine Aktion ausführen. Die Bereitstellung der Funktion durch Ihre Fensterklasse *TMyWindow* könnte dann wie folgt aussehen:

```
class TMyApp : public TApplication {
public:
  TMyApp() : TApplication() {}
  void InitMainWindow();
  int DocMode;
  };

void TMyApp::InitMainWindow() {
  // Geben Sie dmMenu beim Konstruieren von TDocManager nicht an.
  SetDocManager(new TDocManager(dmMDI));
  };

class TMyWindow : public TDecoratedMDIFrame {
public:
  TMyWindow();
  void CmFileOpen();
  /*
     Sie müssen auch die anderen von der Dokumentverwaltung
     bereitgestellten Ereignisbehandlungen zur Verfügung stellen
```

```
  */
  DECLARE_RESPONSE_TABLE(TMyWindow);
  };

DEFINE_RESPONSE_TABLE1(TMyWindow, TDecoratedMDIFrame)
  EV_COMMAND(CM_FILEOPEN, CmFileOpen),
  ...
END_RESPONSE_TABLE;

void TMyWindow::CmFileOpen() {
  // Programmieren Sie hier Ihre weiteren Aufgaben.
  GetApplication()->GetDocManager->CmFileOpen();
  }
```

Erzeugen einer Dokumentklasse

Die Hauptfunktion einer Dokumentklasse ist die Bereitstellung von Callbacks für die angeforderten Datenänderungen in einem View, die Behandlung von Benutzeraktionen, übertragen durch assoziierte Views, und die Mitteilung an assoziierte Views, wenn Daten aktualisiert wurden. *TDocument* gibt den Rahmen für einen Großteil dieser Funktionalität vor. Der Programmierer muß nur die Teile hinzufügen, die für eine spezifische Anwendung des Dokumentmodells benötigt werden.

Konstruieren von TDocument

TDocument ist eine abstrakte Basisklasse, die nicht direkt instanziiert werden kann. Deshalb implementieren Sie Dokumentklassen, indem Sie sie von *TDocument* ableiten.

Das Konstruieren eines *TDocument*-Objektes ist eine einfache Sache: es ist nicht möglich. Sie müssen aber den Konstruktor von *TDocument* aufrufen, wenn Sie eine von *TDocument* abgeleitete Klasse konstruieren. Der Konstruktor von *TDocument* erwartet nur einen Parameter, ein *TDocument* *****, der auf das übergeordnete Dokument des neuen Dokumentes zeigt. Wenn das Dokument kein übergeordnetes Dokument besitzt, können Sie entweder Null übergeben oder keine Parameter übergeben; der Vorgabewert des Parameters ist Null.

Hinzufügen von Funktionalität zu Dokumenten

Sie sollten es grundsätzlich vermeiden, Funktionen von *TDocument* zu überschreiben, die nicht als **virtual** deklariert sind. Die Dokumentverwaltung adressiert alle von *TDocument* abgeleiteten Objekte, als wenn sie aktuell *TDocument*-Objekte wären. Wenn Sie eine nicht-virtuelle Funktion überschreiben, wird sie nicht aufgerufen, wenn die Dokumentverwaltung diese Funktion aufruft. Stattdessen endet die Dokumentverwaltung mit dem Aufruf der *TDocument*-Version der Funktion. Wenn Sie aller-

dings einer virtuelle Funktion überschreiben, ruft die Dokumentverwaltung die korrekte Funktionsversion Ihrer Klasse auf.

Die folgenden Funktionen sind in *TDocument* als **virtual** deklariert:

```
~TDocument                SetDocPath
InStream                  SetTitle
OutStream                 GetProperty
Open                      IsDirty
Close                     IsOpen
Commit                    CanClose
Revert                    AttachStream
RootDocument              DetachStream
```

Sie können diese Funktionen überschreiben, um Ihre eigene Interpretation der Funktion bereitzustellen. Wenn Sie allerdings eine virtuelle Funktion überschreiben, sollten Sie in Erfahrung bringen, was die Basisklassenfunktion ausführt. In Funktionen, bei denen die Basisklassenversion grundlegende Operationen durchführt, sollten Sie auch von Ihrer Version der Funktion aus die Version der Basisklasse aufrufen. Die Basisklassenversionen der meisten Funktionen führen einen Test zur Hierarchie des Dokumentes durch. Dazu gehört auch eine Überprüfung oder eine Benachrichtigung von allen untergeordneten Dokumenten, allen Views, allen offenen Streams usw..

Funktionen für den Datenzugriff

TDocument bietet eine Reihe von Funktionen für den Datenzugriff. Sie können auf Daten wie bei einem einfachen seriellen Stream oder in jeder anderen beliebigen Zugriffart in Ihren abgeleiteten Klassen zugreifen. Die folgenden Abschnitte beschreiben Hilfsfunktionen, die Sie zur Kontrolle verwenden können, wenn das Dokument Zugriffsoperationen auf Daten versucht.

Zugriff auf einen Stream

TDocument enthält zwei Funktionen, *InStream* und *OutStream*, die einen Zeiger auf *TInStream* bzw. *TOutStream* zurückgeben. Die Versionen dieser Funktionen aus *TDocument* geben beide Null zurück, da diese Funktionen keine Aktionen ausführen. Um für Ihre Dokumentklasse einen Zugriff auf Streams zu ermöglichen, müssen Sie diese Funktionen überschreiben, die passende Stream-Klasse konstruieren und einen Zeiger auf ein Stream-Objekt zurückgeben.

TInStream und *TOutStream* sind abstrakte Stream-Klassen, die von *TStream* und *istream* bzw. *ostream* abgeleitet sind. *TStream* bietet eine minimale Funktionalität, um einen Stream mit einem Dokument zu verbinden. *istream* und *ostream* sind Standard-C++ iostreams. Sie müssen Dokumentenspezifische Stream-Klassen von *TInStream* und *TOutStream* ableiten. Die Klassen *TInStream* und *TOutStream* sind im ObjectWindow *Referenzhandbuch* vollständiger dokumentiert. Hier ist nun eine einfache Beschreibung

der Elementfunktionen von *InStream* und *OutStream*. Sowohl der Konstruktor von *TInStream* als auch der von *TOutStream* erwartet zwei Parameter:

```
XXXStream(int mode, LPCSTR strmID = 0)
```

Dabei ist *XXX* entweder *In* oder *Out* und *mode* ist ein Modus zum Öffnen von Streams, der identisch mit den für *istream* und *ostream* verwendeten *open_mode*-Flags ist. *strmID* ist ein Zeiger auf ein existierendes Stream-Objekt. Wenn ein gültiger Zeiger auf ein existierendes Stream-Objekt in *strmID* übergeben wird, wird dieser Stream als das Stream-Objekt des Dokumentes benutzt. Anderenfalls öffnet das Objekt ein neues Stream-Objekt.

Ferner gibt es zwei Funktionen für den Stream-Zugriff mit Namen *AttachStream* und *DetachStream*. Beide Funktionen erwarten eine Referenz auf ein existierendes (d.h. bereits konstruiert und geöffnet), von *TStream* abgeleitetes Objekt. *AttachStream* fügt das von *TStream* abgeleitete Objekt der Liste mit Stream-Objekten des Dokumentes hinzu. Dadurch wird für das Dokument der Zugriff möglich. *DetachStream* durchsucht die Liste mit Stream-Objekten des Dokumentes und löscht das übergebene, von *TStream* abgeleitete Objekt. Beide Funktionen haben geschützten Zugriff und können deshalb nur innerhalb des Dokumentobjektes aufgerufen werden.

Stream-Liste

Jedes Dokument verwaltet eine Liste mit offenen Streams, die aktualisiert wird, wenn Streams hinzugefügt oder gelöscht werden. Diese Liste beginnt mit den *TDocument*-Daten *StreamList*. *StreamList* ist ein *TStrea **, das auf den ersten Stream in der Liste zeigt. Wenn die Liste keine Streams enthält, ist *StreamList* Null. Jedes *TStream*-Objekt in der Liste besitzt ein Element mit Namen *NextStream*, das auf den nächsten Stream in der Liste zeigt.

Wenn ein neuer Stream in einem Dokumentobjekt geöffnet wird oder wenn ein existierender Stream an das Objekt gebunden wird, wird er in die Stream-Liste des Dokumentes hinzugefügt. Wenn ein existierender Stream in einem Dokument geschlossen oder vom Dokument gelöst wird, wird er aus der Stream-Liste des Dokumentes entfernt.

Komplexer Datenzugriff

Streams können nur einfachen seriellen Zugriff auf Daten bieten. In Fällen, in denen ein Dokument Multimedia-Dateien, Datenbanktabellen oder komplexe Daten enthält, benötigen Sie wahrscheinlich anspruchsvollere Zugriffsmethoden. Dazu bietet *TDocument* zwei weitere Zugriffsfunktionen, *Open* und *Close*, die Sie für die Definition Ihres eigenen Verhaltens zum Öffnen und Schließen überschreiben können.

Die *TDocument*-Version von *Open* führt keine Aktionen aus und gibt immer TRUE zurück. Sie können Ihre eigene Version von *Open* für beliebige Zwecke schreiben. Es gibt keine Restriktionen darüber, wie sie das Öffnen eines Dokument definieren. Sie können es so einfach wie möglich oder so komplex wie nötig gestalten. Mit *Open* können Sie ein Dokument öffnen und auch geöffnet halten. Dies ist eine Alternative zum Öffnen des Dokumentes nur bei Bedarf von einem der Stream-Objekte des Dokumentes aus.

Die *TDocument*-Version von *Close* enthält etwas mehr Funktionalität als die von *Open*. Sie überprüft jedes existierende untergeordnete Dokument Ihres Dokumentes und versucht diese zu schließen, bevor Ihr Dokument geschlossen wird. Wenn Sie Ihr eigenes *Close* schreiben, sollten Sie als erstes innerhalb dieser Funktion die *TDocument*-Version von *Close* aufrufen. Dies stellt sicher, daß alle untergeordneten Dokumente berücksichtigt werden, bevor das übergeordnete Dokument schließt. Mit Ausnahme dieser einen Bedingung unterliegt die Implementierung der *Close*-Funktion wie auch bei *Open* keiner weiteren Restriktion. Wie bei *Open* können Sie mit *Close* ein Dokument schließen, wenn es geschlossen werden soll. Dies ist auch wieder eine Alternative zum Schließen des Dokumentes von einem der Stream-Objekte des Dokumentes aus.

Hilfsfunktionen für den Datenzugriff

TDocument bietet auch eine Reihe von Funktionen, die Sie beim Schutz Ihrer Daten verwenden können:

IsDirty überprüft zuerst, ob das Dokument selbst »schmutzig« ist (d.h. es wurde verändert und nicht aktualisiert). Dazu wird der Zustand des Datenelementes *DirtyFlag* abgefragt. Sie überprüft danach, ob irgendwelche untergeordneten Dokumente schmutzig sind, und dann, ob irgendwelche Views schmutzig sind. *IsDirty* gibt TRUE zurück, wenn irgendwelche untergeordneten Dokumente oder Views schmutzig sind.

IsOpen überprüft, ob das Dokument geöffnet gehalten ist oder in seiner Stream-Liste irgendwelche Streams besitzt. Wenn das Dokument nicht offen ist, gibt *IsOpen* FALSE zurück; anderenfalls TRUE.

Commit übergibt im wesentlichen alle Änderungen an Ihren Daten an den Speicher. Wenn Sie *Commit* einmal aufgerufen haben, können Sie keine der vorgenommenen Änderungen mehr rückgängig machen. Die *TDocument*-Version dieser Funktion überprüft alle untergeordneten Dokumente und übergibt ihnen die Verantwortung für deren Änderungen. Wenn irgendein untergeordnetes Dokument FALSE zurückgibt, wird *Commit* abgebrochen und gibt FALSE zurück. Alle untergeordneten Dokumente müssen TRUE zurückgeben, bevor die Funktion *Commit* ihre eigenen Daten aktuell übergibt! Nachdem alle untergeordneten Dokumente TRUE zurückgegeben haben, leert *Commit* alle Views für die Operatio-

nen, die stattgefunden haben könnten, seit das Dokument das letzte Mal die Views überprüft hat. Die Daten in dem Dokument werden entsprechende den Änderungen in den Views aktualisiert und gespeichert. *Commit* gibt dann auch TRUE zurück.

Revert führt die zu *Commit* gegenteilige Funktion aus. Anstelle der Aktualisierung der Änderungen und der Speicherung der Daten löscht *Revert* alle Änderungen, die seit der letzten Übertragung der Daten erfolgten. *Revert* fragt auch alle untergeordneten Dokumente ab und bricht ab, wenn eines der untergeordneten Dokumente FALSE zurückgibt. Wenn alle Operationen erfolgreich sind, gibt *Revert* TRUE zurück.

Schließen eines Dokumentes

Wie bei den meisten anderen Objekten bietet *TDocument* Funktionen, die ein sicheres Schließen und entfernen des Objekte ermöglichen.

~TDocument räumt eine ganze Menge auf. Zuerst entfernt sie ihre untergeordneten Dokumente und schließt alle offenen Streams und andere Zugriffs-Handles. Dann schließt sie der Reihenfolge nach ihr verbundenes Template (aber nur, wenn es sich um das letzte Dokument handelt, das das Template benutzt), schließt alle assoziierten Views, löscht ihre Stream-Liste und entfernt sich schließlich selbst bei ihrem übergeordneten Dokument aus der Liste untergeordneter Dokumente (sofern das Dokument ein übergeordnetes Dokument besitzt) bzw. entfernt sich selbst aus der Dokumentliste der Dokumentverwaltung (sofern das Dokument kein übergeordnetes Dokument besitzt).

Zusätzlich zu einem Destruktor bietet *TDocument* auch eine Funktion *CanClose*, um die Korrektheit für das Schließen sicherzustellen. *CanClose* überprüft zuerst, ob alle ihre untergeordneten Dokumente schließen können. Wenn irgendein untergeordnetes Fenster FALSE zurückgibt, gibt auch *CanClose* FALSE zurück und bricht ab. Wenn alle untergeordneten Dokumente TRUE zurückgeben, ruft *CanClose* die Funktion *FlushDoc* der Dokumentverwaltung auf, die überprüft, ob das Dokument schmutzig ist. Wenn es sauber ist, geben *FlushDoc* und *CanClose* TRUE zurück. Wenn das Dokument schmutzig ist, öffnet *FlushDoc* ein Meldungsfenster, um den Benutzer zur Angabe aufzufordern, ob entweder die Daten gesichert werden sollen, ob die Änderungen verworfen werden sollen oder ob die Operation zum Schließen abgebrochen werden soll.

Erweitern der Dokumentfunktionalität

Die in diesem Abschnitt beschriebenen Funktionen umfassen das Meiste von dem, was Sie für den Aufbau einer funktionierenden Dokumentklasse wissen müssen. Es liegt bei Ihnen, die Funktionalität Ihrer Dokumentklasse zu erweitern. Ihre Klasse benötigt spezielle Funktionen, mit denen Sie Daten manipulieren können, mit denen Sie Informationen verstehen und bearbeiten können, die Sie vom Benutzer durch das zum Dokument

assoziierte View erhalten haben, usw.. Alle diese Funktionalität geht in Ihre von *TDocument* abgeleitete Klasse mit ein.

Auf Grund der Flexibilität des Dok/View-Modells gibt es keine Anforderungen oder Regeln darüber, wie diese Aufgabe zu bewältigen ist. Ein Dokument kann mit praktisch jedem Typ von Daten umgehen, da die Basis für dieses Modell keine außer einer höchst rudimentären Unterstützung für irgendeinen Datentyp enthält. Die in ObjectWindows vorgesehenen Basisklassen stellen nur eine grundlegende Funktionalität zur Verfügung, um Ihre Erweiterungen des Dok/View-Modells zu unterstützen.

Arbeiten mit der Dokumentverwaltung

TDocument enthält zwei Funktionen für den Zugriff auf die Dokumentverwaltung, *GetDocManager* und *SetDocManager*. *GetDocManager* gibt einen Zeiger auf die aktuelle Dokumentverwaltung zurück. Sie können diesen Zeiger dann für den Zugriff auf Daten und Elementfunktionen der Dokumentverwaltung verwenden. *SetDocManager* erlaubt Ihnen, die aktuelle Dokumentverwaltung zu setzen. Alle weitere Funktionalität einer Dokumentverwaltung ist in der Dokumentverwaltung selbst enthalten.

Arbeiten mit Views

TDocument enthält zwei Funktionen für die Arbeit mit Views, *NotifyViews* und *QueryViews*. Beide Funktionen erwarten drei Parameter, ein **int**-Wert, der zu einem Ereignis korrespondiert, ein **long** und ein *TView* *****. Die Bedeutung von dem **long**-Wert hängt vom Ereignis ab und ist im wesentlichen ein Parameter des Ereignisses. Mit *TView* ***** können Sie ein View von Ihrer Abfrage oder Benachrichtigung ausschließen, indem Sie einen Zeiger auf dieses View an die Funktion übergeben. Diese beiden Funktionen sind der Hauptträger für den Austausch von Informationen zwischen Ihrem Dokument und seinen Views.

Beide Funktionen rufen Views durch die Antworttabellen der Views auf. Das allgemeine Makro für Benachrichtigungsbotschaften von ObjectWindows ist EV_OWLNOTIFY. Die Antwortfunktion für EV_OWLNOTIFY-Botschaften haben folgendes Aussehen:

```
BOOL FnName(long);
```

Der von den Funktionsaufrufen von *NotifyViews* oder *QueryViews* verwendete **long**-Wert dient für den **long**-Parameter für die Antwortfunktion.

Sie können *NotifyViews* für eine Nachricht an Ihre untergeordneten Dokumente, an deren assoziierte Views oder an die assoziierten Views des Stammdokumentes verwenden. Diese Nachricht kann besagen, daß sich Daten geändert haben oder daß eine Aktualisierung bzw. irgendein anderes Ereignis, welches sich auf dem Bildschirm niederschlagen muß, ansteht. Die Bedeutung des Ereignisses und des begleitenden Wertes, der

als Ereignisparameter übergeben wurde, sind implementierungsabhängig definiert.

NotifyViews ruft zuerst die *NotifyViews*-Funktionen der zum Dokument untergeordneten Dokumente mit denselben Parametern auf. Wenn alle untergeordneten Dokumente aufgerufen wurden, übergibt *NotifyViews* das Ereignis und den zugehörigen Wert an alle zum Dokument assoziierten Views. *NotifyViews* gibt ein BOOL zurück. Wenn irgendein untergeordnetes Dokument oder irgendein assoziiertes View FALSE zurückgibt, liefert auch *NotifyViews* FALSEE zurück. Anderenfalls gibt *NotifyViews* TRUE zurück.

QueryViews sendet genauso wie *NotifyViews* ein Ereignis und einen begleitenden Parameter. Der Unterschied besteht darin, daß *NotifyViews* TRUE zurückgibt, wenn irgendein untergeordnetes Fenster oder View TRUE zurückgibt, und daß *QueryViews* einen Zeiger auf das erste View, das TRUE zurückgegeben hat, liefert. Dies gibt Ihnen die Möglichkeit, ein View zu finden, das eine Bedingung erfüllt, um anschließend irgendeine Aktion auf diesem View auszuführen. Wenn keines der Views TRUE zurückgibt, liefert *QueryViews* Null zurück.

Ein anderer Unterschied zwischen *NotifyViews* und *QueryViews* besteht darin, daß *NotifyViews* immer das Ereignis und sein Parameter an *alle* untergeordneten Dokumente und assoziierten Fenster schickt. *QueryViews* stoppt dagegen beim ersten View, das TRUE zurückgibt.

Nehmen wir zum Beispiel an, Sie haben eine Dokumentklasse, die Grafikdaten in einer Bitmap enthält. Sie wollen wissen, welches Ihrer assoziierten Views ein bestimmtes Feld der aktuellen Bitmap anzeigt. Sie können ein Ereignis wie beispielsweise WM_CHECKRECT definieren. Dann können sie eine *TRect*-Struktur einrichten, die die Koordinaten des zu überprüfenden Rechteckes enthält. Ein Auszug aus dem dafür zu schreibenden Codes könnte wie folgt aussehen:

```
DEFINE_RESPONSE_TABLE1(TMyView, TView)
  .
  .
  .
  EV_OWLNOTIFY(WM_CHECKREST, EvCheckRest),
  .
  .
  .
END_RESPONSE_TABLE;

void MyDocClass::Function() {

  // Einrichten eines TRect * mit den zu sendenden Koordinaten
  TRect *rect = new TRect(100, 100, 300, 300);

  // QueryViews
  TView *view = QueryViews(WM_CHECKRECT, (long) rect);

  // Lösche alle Änderungen aus dem View
```

```
  if(view)
    view->Clear();
  }

// Die Antwortfunktion des View erhält den Zeiger auf das Rechteck
// Als den long-Parameter zu seiner Antwortfunktion.
BOOL TMyView::EvCheckRest(long item) {
  TRect *rect = (TRect *) item;

  // Überprüfe, ob das Rechteck mit dem des Views übereinstimmt
  if(*rect == this->rect)
    return TRUE;
  else
    return FALSE;
  }
```

Sie können auch Ihre eigenen Ereignismakros einrichten, um View-Benachrichtigungen zu behandeln.

Erzeugen einer View-Klasse

Der Benutzer interagiert fast niemals direkt mit einem Dokument. Stattdessen arbeitet der Benutzer mit einem Oberflächenobjekt wie einem Fenster, einem Dialogfenster oder irgendeiner anderen Darstellungsart, passend für die präsentierten Daten und für die Methode, in der sie präsentiert werden. Dieses Oberflächenobjekt steht allerdings nicht für sich allein. Ein Fenster weiß nichts über die dargestellten Daten, nichts über das Dokument, das diese Daten enthält, und weiß nichts darüber, wie der Benutzer die Daten manipulieren und ändern kann. All diese Funktionalität wird durch das View-Objekt verrichtet.

Ein View-Objekt formt eine Schnittstelle zwischen einem Oberflächenobjekt (welches nur die Aktionen durchführen kann, zu denen es auch beauftragt wurde) und einem Dokumentobjekt (welches keine Informationen darüber besitzt, wie die Oberflächenobjekt mitzuteilen ist, welche Aktionen durchzuführen sind). Die aufgabe des View besteht nun darin, die Lücke zwischen den beiden Objekten zu überbrücken, Daten vom Dokumentobjekt zu lesen und dem Oberflächenobjekt mitzuteilen, wie die Daten anzuzeigen sind.

Dieser Abschnitt geht näher darauf ein, wie eine View-Klasse geschrieben werden muß, damit sie mit Ihren Dokumentklassen zusammenarbeitet.

Konstruieren von TView

Sie können bei *TView* nicht direkt eine Instanz erzeugen. *TView* enthält eine Reihe abstrakter Funktionen und Platzhalterfunktionen, deren Funk-

tionalität in allen abgeleiteten Klassen bereitgestellt werden muß. Sie müssen jedoch den Konstruktor von *TView* aufrufen, wenn Sie Ihr von *TView* abgeleitetes Objekt konstruieren. Der Konstruktor von *TView* erwartet einen Parameter, eine Referenz auf das zum View assoziierte Dokument. Sie müssen eine gültige Referenz auf ein von *TDocument* abgeleitetes Objekt bereitstellen.

Hinzufügen von Funktionalität zu Views

TView besitzt einige abstrakte Funktionen, die Sie in jeder neuen View-Klasse bereitstellen müssen. Sie beinhaltet ebenso einige Platzhalterfunktionen, die keine Basisfunktionalität aufweisen. Sie müssen neue Versionen dieser Funktionen bereitstellen, wenn Sie deren Einsatz planen.

Genauso wie bei *TDocument* sollten Sie eine Funktion von *TView* nicht überschreiben, es sei denn, die Funktion ist virtuell. Wenn Funktionen in *TDocument* Funktionen in Ihrem View aufrufen, adressieren sie das View-Objekt als ein *TView*-Objekt. Wenn Sie eine nicht-virtuelle Funktion überschreiben und das Dokument diese Funktion aufruft, ruft das Dokument die *TView*-Version dieser Funktion auf, wodurch Ihre Funktion in diesem Kontext ungenutzt bleibt.

Virtuelle Funktionen von TView

Die folgenden Funktionen sind als **virtual** deklariert, um Ihnen das Überschreiben zu erlauben und damit eine sinnvolle Funktionalität zur Verfügung zu stellen. Sie sind aber nicht als abstrakt deklariert; er wird von Ihnen nicht *gefordert*, sie für das Konstruieren eines View zu überschreiben. Stattdessen müssen Sie diese Funktionen nur überschreiben, wenn Sie ihren Einsatz planen.

GetViewName gibt den statischen View-Namen zurück. Diese Funktion ais als abstrakte Funktion deklariert. Sie *müssen* eine Definition dieser Funktion in Ihrer View-Klasse bereitstellen.

GetWindow gibt ein *TWindow* * zurück. Dieser Wert sollte das zum View assoziierte Oberflächenobjekt, sofern es eins besitzt, referenzieren. Anderenfalls gibt *GetWindow* Null zurück.

SetDocTitle setzt den Fenstertitel des View. Er sollte gesetzt werden, um die Funktion *SetDocTitle* im Oberflächenobjekt aufzurufen.

Hinzufügen eines Menüs

TView enthält das Datenelement *ViewMenu*, ein *TMenuDescr* *. Sie diesem Element irgendein existierendes *TMenuDescr*-Objekt zuweisen. Das Menü sollte normalerweise im Konstruktor des Views eingerichtet werden. Dieses Menü wird dann mit dem Menü des Rahmenfensters vermischt, wenn das View aktiviert wird.

Hinzufügen einer Anzeige zu einem View

TView sieht von sich aus keine Einrichtungen zum Anzeigen vor – es hat keinen Zeiger auf ein Fenster, keine Grafikfunktionen, keine Funktionen zur Anzeige von Text oder auch keine Tastaturbehandlung. Se müssen diese Funktionalität in Ihren abgeleiteten Klassen bereitstellen. Sie können dazu eine der folgenden Methoden verwenden:

- Fügen Sie einen Zeiger auf ein Oberflächenobjekt in Ihrer abgeleiteten View-Klasse hinzu
- Vermischen Sie die Funktionalität eines Oberflächenobjektes mit der von *TView*, wenn Sie Ihre neue View-Klasse ableiten.

Jede dieser Methoden hat ihre eigenen Vor- und Nachteile. Darauf wird in den nächsten Abschnitte eingegangen. Sie sollten die Pros und Kontras von jedem Ansatz abwägen, bevor Sie entscheiden, wie Sie Ihre View-Klasse aufbauen.

Hinzufügen von Zeigern auf Oberflächenobjekte

Um einen Zeiger auf ein Oberflächenobjekt in Ihrer von *TView* abgeleiteten Klasse hinzuzufügen, fügen Sie nur ein Element in Ihrer neuen Klasse hinzu und instanziieren Sie das Objekt im Konstruktor der View-Klasse. Der Zugriff auf die Daten und Elementfunktionen des Oberflächenobjektes erfolgt immer über den Zeiger.

Der Vorteil dieser Methode besteht darin, daß Sie verschiedene Oberflächenelemente auf einfache Weise zuordnen und entfernen können. Sie können ebenso verschiedene Typen von Oberflächenobjekten verwenden. Dazu müssen Sie den Zeiger nur als Zeiger auf eine für alle verschiedenen Oberflächenobjekten gemeinsame Basisklasse deklarieren. Sie können zum Beispiel die meisten Typen von Oberflächenobjekten verwenden, indem Sie den Zeiger als *TWindow* * deklarieren.

Der Nachteil dieser Methode bestehet darin, daß die Ereignisbehandlung zuerst entweder über das Oberflächenobjekt oder über die Anwendung muß. Damit sind Sie entweder dazu gezwungen, eine abgeleitete Klasse für das Oberflächenobjekt zu verwenden, um Ihre eigenen Funktionen für die Ereignisbehandlung, die das View-Objekt referenzieren, hinzuzufügen, oder aber Sie behandeln Ereignisse über das Anwendungsobjekt. Bei jeder Alternative verringern Sie Ihre Flexibilität bei der Behandlung von Ereignissen.

Vermischen von TView mit Oberflächenobjekten

Das Vermischen von *TView* bzw. eines von *TView* abgeleiteten Objektes mit einer Oberflächenelementklasse ermöglicht Ihnen die Anzeige von Daten aus einem Dokument und erlaubt Ihnen dadurch auch die vollständige Behandlung des Datenflusses zum und vom Dokumentobjekt. Das Vermischen einer View-Klasse mit einer Oberflächenobjektklasse ist eine einfache Aufgabe, die allerdings mit Sorgfalt durchgeführt werden muß.

Um einen neue Klasse abzuleiten, können Sie die Klasse einfach basierend auf Ihrer View-Basisklasse (*TView* oder eine von *TView* abgeleitete Klasse) und auf dem ausgewählten Oberflächenobjekt definieren. Der neue Konstruktor sollte die Konstruktoren beider Basisklassen aufrufen sowie alle Daten initialisieren, die eine Einstellung benötigen. Die neue Klasse muß mindestens alle Funktionen definieren, die in den Basisklassen als abstrakte Funktionen deklariert sind. Ebenso sollte sie einerseits die benötigten Funktionen für die speziellen Bildschirmaktivitäten definieren, andererseits auch die Funktionen zur Ereignisbehandlung bereitstellen, die für die Kommunikation sowohl mit dem Oberflächenelement als auch mit dem Dokumentobjekt benötigt werden.

Der Vorteil diese Ansatzes besteht darin, daß das resultierende View in sich geschlossen ist. Die Behandlung von Ereignissen wird an zentraler Stelle durchgeführt. Dadurch wird der Bedarf an Botschaftsbehandlung auf dem Niveau der Anwendung reduziert. Die Kontrolle der Oberflächenelemente erfolgt nicht über einen Zeiger, sie ist aber auch in die neue View-Klasse integriert.

Wenn Sie allerdings diesen Ansatz verwenden, verlieren Sie die Flexibilität, die man bei einem Zeiger besitzt. Sie können nicht so schnell neue Oberflächenobjekte zuordnen und entfernen; das Oberflächenobjekt ist ein organischer Bestandteil des ganzen View-Objektes. Ebenso können Sie nicht verschiedene Objekttypen austauschen, indem Sie einen Basiszeiger auf die verschiedenen Oberflächenobjektklassen benutzen. Ihre neue View-Klasse ist fest mit einem einzelnen Oberflächenelement verbunden.

Schließen eines View

Wie bei den meisten anderen Objekten, bietet *TView* Funktionen für ein sicheres Schließen und Entfernen des Objektes.

~TView führt nur sehr wenig durch. Sie ruft die *DetachView*-Funktion ihres assoziierten Dokumentes auf, um sich selbst aus der Liste der Views im Dokument zu entfernen.

TView stellt auch eine Funktion *CanClose* bereit. Die Funktion *CanClose* ruft die *CanClose*-Funktion ihres assoziierten Dokumentes auf. Deshalb hängt die Möglichkeit des View zum Schließen vollständig von der Möglichkeit des Dokumentes zum Schließen ab.

Dok/View-Ereignisbehandlung

Sie sollten normalerweise Dok/View-Ereignisse sowohl über das Anwendungsobjekt als auch über des Oberflächenelement Ihres View behandeln. Sie können entweder die Anzeige eines View über einen Zeiger auf

ein Oberflächenobjekt kontrollieren oder die Funktionalität des Oberflächenobjektes mit einer View-Klasse vermischen (siehe Konstruieren eines Oberflächenelementes).

Weitere Informationen über Ereignisbehandlung und Antworttabellen in einer ObjectWindows-Anwendung findet sich in Kapitel 5.

Dok/View-Ereignisbehandlung im Anwendungsobjekt

Das Anwendungsobjekt behandelt im allgemeinen nur ein paar Ereignisse, die anzeigen, wenn ein Dokument oder ein View erzeugt oder entfernt wurde. Das Ereignis *dnCreate* wird immer dann hinterlegt, wenn ein View oder Dokument erzeugt wird. Das Ereignis *dnClose* wird immer dann hinterlegt, wenn ein View oder Dokument entfernt wird.

Um Antworttabelleneinträge für diese Ereignisse einzurichten, können Sie die Makros EV_OWLDOCUMENT und EV_OWLVIEW in Ihrer Antworttabelle hinzufügen:

1. Verwenden Sie das Makro EV_OWLDOCUMENT, um folgendes zu testen:
2. Das Ereignis *dnCreate*, wenn ein neues Dokument erzeugt wird. Der für die Behandlungsfunktion verwendete Standardname ist *EvNewDocument*. *EvNewDocument* erwartet eine Referenz auf das neue, von *TDocument* abgeleitete Objekt und gibt **void** zurück.
3. Das Ereignis *dnClose*, wenn ein Dokumentobjekt entfernt werden soll. Der für die Behandlungsfunktion verwendete Standardname ist *EvCloseDocument*. *EvCloseDocument* erwartet eine Referenz auf das von *TDocument* abgeleitete Objekt, das entfernt wird, und gibt **void** zurück.

Die Antworttabelleneinträge und Funktionsdeklarationen für diese beiden Makros würden wie folgt aussehen:

```
DEFINE_RESPONSE_TABLE1(MyDVApp, TApplication)
  ⋮
  EV_OWLDOCUMENT(dnCreate, EvNewDocument),
  EV_OWLDOCUMENT(dnClose, EvCloseDocument),
  ⋮

END_RESPONSE_TABLE;

void EvNewDocument(TDocument& document);
void EvCloseDocument(TDocument& document);
```

- Verwenden Sie das Makro EV_OWLVIEW, um folgendes zu testen:

  Das Ereignis *dnCreate*, wenn ein neues View-Objekt konstruiert wird. Der für die Behandlungsfunktion verwendete Standardname ist *EvNewView*. *EvNewView* erwartet eine Referenz auf das neue, von *TView* abgeleitete Objekt und gibt **void** zurück.

Wenn das View ein Fenster-Oberflächenelement enthält (entweder durch Vererbung oder über einen Zeiger) ist das Oberflächenelement typischerweise noch nicht erzeugt, wenn das View konstruiert wird.

Sie können dann die Erzeugungsattribute des Oberflächenelementes verändern, bevor die Funktion *Create* wirklich aufgerufen wird.

Das Ereignis *dnClose*, wenn ein View-Objekt entfernt wird. Der für die Behandlungsfunktion verwendete Standardname ist *EvCloseView*. *EvCloseView* erwartet eine Referenz auf das von *TView* abgeleitete Objekt, das entfernt wird, und gibt **void** zurück.

Die Antworttabelleneinträge und Funktionsdeklarationen für diese beiden Makros würden wie folgt aussehen:

```
DEFINE_RESPONSE_TABLE1(MyDVApp, TApplication)
  .
  .
  .
  EV_OWLVIEW(dnCreate, EvNewView),
  EV_OWLVIEW(dnClose,  EvCloseView),
  .
  .
  .

END_RESPONSE_TABLE;

void EvNewView(TView &view);
void EvCloseView(TView &view);
```

Dok/View-Ereignisbehandlung in einem View

Die Header-Datei docview.h bietet eine Reihe von Antworttabellenmakros für vordefinierte Ereignisse sowie die Namen der Behandlungsfunktionen und eine Typüberprüfung für die Funktionsdeklarationen. Sie können auch Ihre eigenen Ereignisse und Funktionen definieren, um diese Ereignisse mit den Makros NOTIFY_SIG und VN_DEFINE zu behandeln.

Behandlung vordefinierter Dok/View-Ereignisse

Es gibt eine Reihe vordefinierter Dok/View-Ereignisse. Zu jedem Ereignis ist ein korrespondierendes Antworttabellenmakro und eine korrespondierender Kopf einer Behandlungsfunktion definiert. Beachten Sie, daß das Dok/View-Modell keine vollständigen Versionen dieser Funktionen bereitstellt. Sie müssen die Funktionen in Ihrer View-Klasse deklarieren und die passende Funktionalität für jede Funktion bereitstellen.

Tabelle 11.2: Vordefinierte Dok/View-Ereignisbehandlungen

Antworttabellenmakro	Ereignisname	Ereignisbehandlung	Ereignis
EV_VN_VIEWOPENED	*vnViewOpened*	*VnViewOpened*(*TView**)	Zeigt an, daß ein neues View konstruiert wurde.
EV_VN_VIEWCLOSED	*vnViewClosed*	*VnViewClosed*(*TView**)	Zeigt an, daß ein View entfernt wird.
EV_VN_DOCOPENED	*vnDocOpened*	*VnDocOpened*(**int**)	Zeigt an, daß ein neues Dokument geöffnet wurde.
EV_VN_DOCCLOSED	*vnDocClosed*	*VnDocClosed*(**int**)	Zeigt an, daß ein Dokument geschlossen wurde.
EV_VN_COMMIT	*vnCommit*	*VnCommit*(BOOL)	Zeigt an, daß die an den Daten im View vorge-

			nommenen Änderungen an das Dokument übergeben werden sollten.
EV_VN_REVERT	*vnRevert*	*VnRevert*(BOOL)	Zeigt an, daß die an den Daten im View vorgenommenen Änderungen verworfen und die Daten aus dem Dokument wieder hergestellt werden sollen.
EV_VN_ISDIRTY	*vnIsDirty*	*VnIsDirty*(**void**)	Sollte TRUE zurückgeben, wenn Änderungen an den Daten im View vorgenommen und noch nicht an das Dokument übergeben wurden, anderenfalls FALSE.
EV_VN_ISWINDOW	*vnIsWindow*	*VnIsWindow*(HWND)	Sollte TRUE zurückgeben, wenn der HWND-Parameter derselbe ist wie der des Anzeigefensters für das View

Alle für diese Botschaften verwendeten Ereignisbehandlungsfunktionen geben einen BOOL-Wert zurück.

Hinzufügen eigener View-Ereignisse

Sie können die Makros VN_DEFINE und NOTIFY_SIG verwenden, um Ihre eigenen selbstdefinierten View-Ereignisse zu hinterlegen und um korrespondierende Antworttabellenmakros und Ereignisbehandlungsfunktionen zu definieren. Dieser Abschnitt beschreibt, wie ein Ereignis definiert wird und wie die Ereignisbehandlungsfunktion und das Antworttabellenmakro für dieses Ereignis einzurichten ist.

Zuerst müssen Sie den Namen des Ereignisses, das Sie behandeln wollen, definieren. Nach Konvention sollte dieser Name mit den Buchstaben *vn* beginnen, gefolgt vom Ereignisnamen. Ein selbstdefiniertes View-Ereignis sollte als ein **const int** mit einem Wert größer als *vnCustomBase* definiert sein. Sie können Ihre Ereigniswerte mit *vnCustomBase* plus einem Offset-Wert definieren. Nehmen wir zum Beispiel an, Sie definieren ein Ereignis mit Namen *vnPenChange*. Der Code würde in etwa wie folgt aussehen:

```
const int vnPenChange = vnCustomBase + 1;
```

Benutzen Sie als nächstes das Makro NOTIFY_SIG, um den Kopf der Ereignisbehandlungsfunktion anzugeben. Das Makro NOTIFY_SIG erwartet zwei Parameter, als ersten den Ereignisnamen und als zweiten den exakten Parametertyp, der an die Funktion übergeben wird. Die Größe dieses Parameters kann nicht größer als Typ **long** sein; wenn das übergebene Objekt größer als ein **long** ist, müssen Sie einen Zeiger übergeben. Nehmen wie für das *vnPenChange*-Ereignis zum Beispiel an, Sie wollen ein *TPen*-Objekt an die Ereignisbehandlungsfunktion übergeben. Da ein *TPen*-Objekt ein ganzes Stück größer als ein **long** ist, müssen Sie das Objekt mittels eines Zeigers übergeben. Das Makro würde in etwa wie folgt aussehen:

```
NOTIFY_SIG(vnPenChange, TPen *)
```

Nun müssen Sie das Antworttabellenmakro für Ihr Ereignis definieren. Nach Konvention benutzt der Makroname den Ereignisnamen in Großbuchstaben, wobei »EV_VN_« vorangeht. Verwenden Sie das Makro **#define**, um den Makronamen zu definieren. Verwenden Sie das Makro VN_DEFINE, um das Makro selbst zu definieren. Dieses Makro erwartet drei Parameter:

- Ereignisname
- Name der Ereignisbehandlungsfunktion (nach Konvention dasselbe wie der Ereignisname, wobei *Vn* anstelle eines *vn* wie beim Ereignisnamen vorangeht)
- Größe des Parameters für die Ereignisbehandlungsfunktion; diese Angabe kann vier verschiedene Werte annehmen:

  void

  int (Größe eines int-Parameters hängt von der Umgebung ab)

  long (32-Bit-Ganzzahl oder far-Zeiger)

  Zeiger (Größe eines Zeiger-Parameters hängt vom Speichermodell ab)

Sie sollten denjenigen Wert angeben, der am ehesten mit dem Parametertyp der Ereignisbehandlungsfunktion korrespondiert.

Die Definition des Antworttabellenmakro für das *vnPenChange*-Ereignis würde in etwa wie folgt aussehen:

```
#define EV_VN_PENCHANGE \
  VN_DEFINE(vnPenChange, VnPenChange, pointer)
```

Beachten Sie, daß der dritte Parameter des Makros VN_DEFINE in diesem Fall pointer ist. Dies zeigt die Größe des Wertes an, der an die Ereignisbehandlungsfunktion übergeben wird.

Dok/View-Eigenschaften

Jedes Dokument- und View-Objekt enthält eine Liste mit Eigenschaften, gepaart mit Funktionen, die Sie für die Abfrage und Änderung dieser Eigenschaften verwenden können. Die Eigenschaften enthalten Informationen über das Objekt und seine Fähigkeiten. Wenn die Dokumentverwaltung ein Dokument- oder View-Objekt erzeugt oder entfernt, sendet sie ein Benachrichtigungsereignis an die Anwendung. Die Anwendung kann die Eigenschaften des Objektes abfragen, um über das weitere Vorgehen zu entscheiden. Views können ebenso auf die Eigenschaften ihres assoziierten Dokumentes zugreifen.

Eigenschaftswerte und -namen

TDocument und *TView* besitzen jeweils einige allgemeine Eigenschaften. Diese Eigenschaften sind in allen von *TDocument* und *TView* abgeleiteten Klassen verfügbar. Diese Eigenschaften sind durch eine Liste mit aufgezählten Werten indiziert. Die erste Eigenschaft für jede von *TDocument* und *TView* abgeleitete Klasse sollte *PrevProperty* sein. Der letzte Wert in der Eigenschaftsliste sollte *NextProperty* sein. Diese beiden Werte begrenzen die Eigenschaftsliste von jedem Dokument- und View-Objekt. Sie stellen sicher, daß Ihre Eigenschafstliste mit den korrekten Wert startet und nicht den Wert einer anderen Eigenschaft überschreitet. Ferner wird damit für abgeleitete Klassen sicher gestellt, daß Eigenschaftslisten mit einem passenden Wert starten. *PrevProperty* sollte auf den Wert *NextProperty* - 1 der direktesten Basisklasse gesetzt werden.

Eine Eigenschafstliste für eine von *TDocument* abgeleitet Klasse müßte zum Beispiel wie folgt aussehen:

```
enum {
  PrevProperty = TDocument::NextProperty-1,
  Size,
  StorageSize,
 NextProperty,
};
```

Beachten Sie Verwendung des Zugriffsoperators (**::**) beim Setzen von *PrevProperty*. Dies stellt sicher, daß Sie *PrevProperty* auf den korrekten Wert für *NextProperty* setzen.

Eigenschaftsnamen sind normalerwiese in einem Array mit Strings enthalten. Dabei korrespondiert die Position von jedem Namen im Array mit seinem aufgezählten Eigenschaftsindex. Wenn Sie allerdings Eigenschaften einer abgeleiteten Klasse hinzufügen, können Sie die Strings auf eine beliebige Art speichern und zugreifen. Da Sie die Funktionen für den Zugriff auf die Eigenschaften selbst schreiben müssen, sind komplizierte Speicherschemata nicht empfohlen. Ein Eigenschaftsname sollte eine einfache Beschreibung der Eigenschaft sein.

Eigenschaftsattribute sind ebenfalls normalerwiese in einem Array enthalten, dieses Mal in einem Array mit **int**-Werten. Sie können dies wiederum nach Ihren Vorstellungen behandeln. Die übliche Praktik sieht allerdings vor, daß die Attribute für eine Eigenschaft in einem Array korrespondierend mit dem Wert des Eigenschaftsindex liegen. Die Attribute zeigen an, wie auf die Eigenschaft zugegriffen werden kann:

Tabelle 11.3: Dok/View-Eigenschaftsattribute

Attribut	Funktion
pfGetText	Eigenschaft als Textformat zugreifbar
pfGetBinary	Eigenschaft als ursprüngliches nicht-Text-Format zugreifbar
pfConstant	Eigenschaft kann nicht geändert werden, wenn das Objekt einmal erzeugt ist.

pfSettable	Eigenschaft setztbar, muß ursprüngliches Format unterstützen.
pfUnknown	Eigenschaft definiert, aber nicht verfügbar in **diesem** Objekt.
pfHidden	Eigenschaft sollte vor normalem Durchsehen versteckt werden (Der Benutzer soll deren Name und Wert nicht sehen)
pfUserDef	Eigenschaft wurde zur Laufzeit vom Benutzer definiert.

Zugreifen auf Eigenschaftsinformationen

Es gibt eine Reihe von Funktionen, die sowohl in *TDocument* als auch in *TView* für den Zugriff auf Eigenschaftsinformationen des Dok/View-Objektes verwendet werden. Alle diese Funktionen sind als virtuelle Funktionen deklariert. Da die Eigenschafts-Zugriffsfunktionen virtuell sind, wird die Funktion in der am weitesten abgeleiteten Klasse zuerst aufgerufen und kann damit Eigenschaften, die in der Basisklasse definiert sind überschreiben. Es liegt in der Verantwortung jeder Klasse, den Eigenschaftszugriff zu implementieren und dessen Eigenschaftsnamen aufzulösen.

Normalerweise greifen Sie auf eine Eigenschaft über ihre Indexnummer zu. Verwenden Sie die Funktion *FindProperty* mit dem Eigenschaftsnamen. *FindProperty* erwartet ein **char *** als Parameter und durchsucht die Eigenschaftsliste nach einer Eigenschaft mit demselben Namen. Sie gibt einen **int**-Wert zurück, welcher für nachfolgende Aufrufe als Eigenschaftsindex verwendet wird.

Sie können ferner die Funktion *PropertyName* verwenden, um den Eigenschaftsnamen zu einem Index herauszufinden. *PropertyName* erwartet ein **int** als Parameter und gibt ein **char ***-Wert zurück, der den Namen der Eigenschaft enthält.

Sie können die Attribute einer Eigenschaft über die Funktion *PropertyFlags* ermitteln. Diese Funktion erwartet einen **int**-Wert als Parameter, welcher der Index der gewünschten Eigenschaft sein sollte, und gibt ein **int** zurück. Sie können bestimmten, ob ein Flag gesetzt ist, indem Sie den Operator **&** verwenden. Um zum Beispiel zu bestimmen, ob Sie den Eigenschaftswert in Textform abfragen können, sollten Sie überprüfen, ob das Flag *pfGetText* gesetzt ist:

```
if(doc->PropertyFlags() & pfGetText) {
  // Ermittle Eigenschaft als Text....
}
```

Ermitteln und Setzen von Eigenschaften

Sie können die Funktionen *GetProperty* und *SetProperty* verwenden, um die Werte der Eigenschaften eines Dok/View-Objektes abzufragen oder zu verändern.

Die Funktion *GetProperty* ermittelt den Wert einer Eigenschaft:

```
int GetProperty(int index, void far* dest, int textlen = 0);
```

Dabei gilt folgendes:

- *index* ist der Eigenschaftsindex.
- dest wird von *GetProperty* verwendet, um die Eigenschaftsdaten aufzunehmen.
- textlen zeigt die Größe des Speicher-Array an, auf das *dest* zeigt. Wenn *testlen* Null ist, werden die Eigenschaftsdaten in binärer Form und anderenfalls in Textform zurückgegeben. Daten können nur dann in binärer Form zurückgegeben werden, wenn Attribut *pfGetBinary* gesetzt ist. Sie können nur dann in Textform zurückgegeben werden, wenn das Attribut *pfGetText* gesetzt ist. Um die binären Daten von Eigenschaften abzufragen und zu setzen, müssen der Datentyp und die Bedeutungen dem Aufrufer bekannt sein.

Die Funktion *SetProperty* setzt den Wert einer Eigenschaft:

```
BOOL SetProperty(int index, const void far* src)
```

Dabei gilt folgendes:

- *index* ist der Eigenschaftsindex.
- src enthält die Daten, auf welche die Eigenschaft gesetzt werden soll. *src* muß im korrekten ursprünglichen Format für die Eigenschaft vorliegen.

Eine abgeleitete Klasse, die Eigenschaftsnamen dupliziert, sollte dasselbe Verhalten und denselben Datentyp bereitstellen.

K A P I T E L 12

Dialogelementobjekte

Windows bietet eine Reihe von Standardelementen für die Benutzeroberfläche, die als *Dialogelemente* bezeichnet werden. Dialogelemente sind Fenster mit spezialisiertem Verhalten. ObjectWindows bietet mehrere spezialisierte Dialogelemente, die *spezielle Dialogelemente* genannt werden. ObjectWindows stellt Oberflächenobjekte für die Dialogelemente bereit, so daß Sie diese Dialogelemente in Ihren Anwendungen verwenden können. Die Oberflächenobjekte für die Dialogelemente werden als *Dialogelementobjekte* bezeichnet.

Siehe Kapitel 4 für weitere Informationen zu Oberflächenobjekten

Dieses Kapitel umfaßt die folgenden Themen:

- Allen Dialogelementobjekten gemeinsame Aufgaben
- Konstruieren und Entfernen von Dialogelementobjekten
- Kommunizieren mit Dialogelementobjekten

Verwenden der verschiedenen Dialogelementobjekte
Einstellen und Abfragen von Dialogelementwerten

Dialogelementklassen

Die untenstehende Tabelle listet alle von ObjectWindows zur Verfügung gestellte Dialogelementklassen auf.

Tabelle 12.1: Dialogelemente und ihre ObjectWindows-Klassen

Dialogelement	Klassenname	Beschreibung
Standarddialogelemente von Windows		
Listenfeld	*TListBox*	Eine Listenfeld mit auswählbaren Einträgen
Bildlaufleiste	*TScrollBar*	Eine Bildlaufleiste ähnlich denen in bildlauffähigen Fenstern und Listenfeld.
Schalter	*TButton*	Ein Schalter mit zugehörigem Text.
Markierungsfeld	*TCheckBox*	Ein Schalter, der aus einem zugehörigen Text und einem Feld besteht, das markiert (an) oder nicht-markiert (aus) sein kann.

Schaltfeld	*TRadioButton*	Ein Schalter, der markiert (an) oder nicht-markiert (aus) sein kann. Er wird normalerweise in Gruppen mit mehreren Schaltfeldern verwendet, die sich gegenseitig ausschließen.
Gruppenfeld	*TGroupBox*	Ein statisches Rechteck, optional mit Text in der oberen linken Ecke.
Editierfeld	*TEdit*	Ein Feld für den Benutzer zur Eingabe von Text.
Statisches Element	*TStatic*	Sichtbarer Text, den der Benutzer nicht ändern kann.
Kombinationsfenster	*TComboBox*	Ein kombiniertes Listenfeld und Editierfeld oder statisches Element.

Spezielle Dialogelemente von ObjectWindows

Gleitregler	*THSlider* und *TVSlider*	Horizontale und vertikale Dialogelemente. bei denen der Benutzer zwischen einem oberen und unteren Grenzwert auswählen kann (ähnlich den Bildlaufleisten)
Meßanzeige	*TGauge*	Statische Dialogelemente, die den Grad einer Prozeßvervollständigung anzeigen.

Beachten Sie, daß Beispielprogramme zu Dialogelementobjekten in \BORLANDC\EXAMPLES\OWL\OWLAPPS zu finden sind.

Was sind Dialogelementobjekte?

Für Windows sind Dialogelemente lediglich spezialisierte Arten von Fenstern. In ObjectWindows ist *TControl* von *TWindow* abgeleitet, wodurch viele Ähnlichkeiten bei der Ihrer Verwendung von Dialogelementobjekten mit der von beliebigen anderen Fensterobjekten auftreten. Dialogelementobjekte sind den Fensterobjekten ähnlich, wenn es sich um das Erzeugen und Entfernen der Objekte oder um das Verhalten als untergeordnetes Fenster handelt. Standarddialogelemente unterscheiden sich von anderen Fenstern darin, daß Windows deren Ereignisbotschaften behandelt und daß Windows für deren Darstellung verantwortlich ist. Die speziellen Dialogelemente von ObjectWindows übernehmen diese Darstellungen selbst. Die Dialogelementklassen von ObjectWindows enthalten den Code, der für die Darstellung der Dialogelemente und für Behandlung von Ereignissen erforderlich ist.

In vielen Fällen können Sie direkt die Instanzen der oben aufgelisteten Klassen verwenden. Manchmal kann aber auch die Erzeugung abgeleiteter Klassen für ein spezielles Verhalten nötig sein. Sie könnten zum Beispiel eine spezielle Listenfeld-Klasse, abgeleitet von *TListBox*, mit Namen *TFontListBox* benötigen, die die Namen aller in Ihrer Anwendung verwendbaren Schriften aufnimmt und sie automatisch anzeigt, wenn Sie eine Instanz der Klasse erzeugen.

Konstruieren und Entfernen von Dialogelementobjekten

Unabhängig davon, welche Art von Dialogelementobjekt Sie verwenden, gibt es für alle eine Reihe gleicher Aufgaben zu erledigen:

- Konstruieren des Dialogelementobjektes
- Anzeigen des Dialogelementes
- Entfernen des Dialogelementes

Konstruieren von Dialogelementobjekten

Das Konstruieren eines Dialogelementobjektes unterscheidet sich nicht von dem Konstruieren irgendeines anderen untergeordneten Fensters. Generell ruft der Konstruktor eines übergeordneten Fensters die Konstruktoren aller seiner untergeordneten Fenster auf. Dialogelemente kommunizieren mit übergeordneten Fenstern zusätzlich zu den üblichen Verbindungen zwischen unter- und übergeordnetem Fenster auf eine spezielle Weise (Benachrichtigung genannt).

Benachrichtigungen sind in Kapitel 4 beschreiben

Um ein Dialogelementobjekt zu konstruieren und zu initialisieren, gehen Sie wie folgt vor:

- Fügen Sie einen Zeiger auf ein Dialogelementobjekt als Datenelement beim übergeordneten Fenster hinzu
- Rufen Sie den Konstruktor des Dialogelementobjektes auf
- Ändern Sie Attribute des Dialogelementes
- Initialisieren Sie das Dialogelement in *SetupWindow*

Jeder dieser Schritte wird in den folgenden Abschnitten beschrieben.

Hinzufügen des Zeigers auf ein Dialogelementobjekt als Datenelement

Wenn Sie ein Dialogelement in einem Fenster konstruieren, wollen Sie oftmals einen Zeiger auf das Dialogelement im Datenelement des Fensterobjektes festhalten. Dies ist zum Vorteil des Zugriffes auf Elementfunktionen des Dialogelementes. Hier ist ein Fragment eines übergeordneten Fensterobjektes mit der Deklaration eines Zeigers auf einen Schalter als Dialogelementobjekt:

```
class TMyWindow : public TWindow {
  TButton *OkButton;
  .
  .
  .
  };
```

Dialogelemente, die Sie selten manipulieren, wie beispielsweise statischer Text oder Gruppenfelder benötigen diese Zeiger-Datenelemente nicht. Das folgende Beispiel konstruiert ein Gruppenfeld ohne ein Datenelement und einen Schalter mit einem Datenelement (*OkButton*):

```
TMyWindow::TMyWindow(TWindow *parent, const char far *title)
          :TWindow(parent, title)
```

```
{
  new TGroupBox(this, ID_GROUPBOX, "Group box", 10, 10, 100, 100);
  OkButton = new TButton(this, IDOK, "OK", 10, 200, 50, 50, TRUE);
}
```

Aufrufen des Konstruktors für Dialogelementobjekte

Der Konstruktor eines Dialogelementobjektes hat mindestens sieben Parameter:

- Ein Zeiger auf das übergeordnete Fensterobjekt
- Eine Ressource-ID
- Die x-Koordinate der linken oberen Ecke
- Die y-Koordinate der linken oberen Ecke
- Die Breite
- Die Höhe
- Bibliotheks-ID (optional)

Einer der Konstruktoren von *TListBox* ist zum Beispiel wie folgt deklariert:

```
TListBox(TWindow *parent, int id,
         int x, int y, int w, int h,
         TLibID libID = 0);
```

Ferner existieren Konstruktoren für die Assoziation eines Dialogelementobjektes mit einem Oberflächenelement (zum Beispiel einem Dialogfenster), das von einer Ressourcen-Definition aus erzeugt wurde:

```
TListBox(TWindow* parent, int resourceId, TModule* module = 0);
```

Ändern der Attribute von Dialogelementen

Alle Dialogelementobjekte erhalten die Standard-Fenstereigenschaften WS_CHILD, WS_VISIBLE, WS_GROUP und WS_TABSTOP. Wenn Sie die Art eines Dialogelementes ändern wollen, müssen Sie sein Element *Attr.Style* manipulieren, wie es in Kapitel 14 beschrieben ist. Jeder Dialogelementtyp hat ferner andere Attribute, die dessen spezielle Eigenschaften definieren.

Jedes Dialogelementobjekt erbt bestimmte Fenstereigenschaften, von seinen Basisklassen. Sie sollten nur selten einen Wert an *Attr.Style* zuweisen. Stattdessen sollten Sie die bitweisen Zuweisungsoperatoren (**|=** und **&=**)verwenden, um die gewünschte Fenstereigenschaft »einzumaskieren« oder »auszumaskieren«. Ein Beispiel:

```
// Einmaskieren der Fenstereigenschaft WS_BORDER
Attr.Style |= WS_BORDER;

// Ausmaskieren der Fenstereigenschaft WS_VSCROLL
Attr.Style &= ~WS_VSCROLL;
```

Die Verwendung der bitweisen Zuweisungsoperatoren stellt sicher, daß Sie keine Eigenschaft versehentlich entfernen.

Initialisieren das Dialogelementes

Das Oberflächenelement eines Dialogelementobjektes wird automatisch durch die Elementfunktion *SetupWindow* erzeugt, die durch das übergeordnete Fensterobjekt geerbt wurde. Stellen Sie sicher, daß Sie bei einer neu abgeleiteten Fensterklasse die Elementfunktion *SetupWindow* der Basisklasse aufrufen, bevor Sie eine Manipulation ihrer Dialogelemente versuchen (z.B. durch Aufruf von Elementfunktionen des Dialogelementobjektes, durch Senden von Botschaften an diese Dialogelemente usw.).

Sie dürfen Dialogelemente im Konstruktor ihres übergeordneten Fensterobjektes initialisieren. Zu diesem Zeitpunkt wurden die Oberflächenelemente der Dialogelemente noch nicht erzeugt.

Hier ist ein typisches Aussehen von *SetupWindow*:

```
void TMyWindow::SetupWindow()
{
  TWindow::SetupWindow();    // erzeugt untergeordnete Dialogelemente

  list1->AddString("Item 1");
  list1->AddString("Item 2");
}
```

Anzeigen von Dialogelementen

Es ist für die Anzeige von Dialogelementen nicht notwendig, die Windows-Funktion *Show* aufzurufen. Dialogelemente sind untergeordnete Fenster und Windows zeigt sie automatisch an und stellt sie zusammen mit dem übergeordneten Fenster automatisch dar. Sie können aber *Show* dazu verwenden, um Dialogelemente bei Bedarf zu verstecken oder zu offenbaren.

Entfernen des Dialogelementes

Entfernen von Dialogelementen liegt in der Verantwortung des übergeordneten Fensters. Das Oberflächenelement des Dialogelementes wird zusammen mit dem übergeordneten Fenster automatisch entfernt, wenn der Benutzer das Fenster oder die Anwendung schließt. Der Destruktor des übergeordneten Fensters entfernt automatisch seine untergeordneten Fensterobjekte (einschließlich untergeordneter Dialogelementobjekte).

Kommunizieren mit Dialogelementobjekten

Die Kommunikation zwischen einem Fensterobjekt und seinen Dialogelementobjekten ist auf bestimmte Weise der Kommunikation zwischen einem Dialogfensterobjekt und dessen Dialogelementen ähnlich. Wie bei einem Dialogfenster benötigt auch ein Fenster einen Mechanismus für die Manipulation seiner Dialogelemente und für Antwort auf ein Dialogelementereignis wie die Auswahl in einem Listenfeld.

Manipulieren von Dialogelementen

Ein Weg, auf dem Dialogfenster ihrer Dialogelemente manipulieren, besteht in dem Senden von Botschaften an die Elemente. Dies erfolgt über Elementfunktionen, die von *TWindow* geerbt wurden, zusammen mit einer Dialogelementbotschaft wie LB_ADDSTRING. Dialogelementobjekte vereinfachen diesen Prozeß erheblich, indem sie Elementfunktionen bereitstellen, die die Dialogelementbotschaften für Sie senden. *TListBox::AddString* zum Beispiel erwartet einen String als Parameter und fügt ihn dem Listenfeld hinzu, indem sie die Elementfunktion *HandleMessage* des Listenfeldobjektes aufruft:

```
int TListBox::AddString(const char far* str)
{
  return (int)HandleMessage(LB_ADDSTRING, 0, (LPARAM)str);
}
```

Ein Aufruf der Elementfunktionen des Dialogelementobjektes stellt sich einfach dar:

```
listBox1->AddString("Atlantic City"); // Dabei ist listBox1 eine TListBox *
```

Antworten auf Dialogelemente

Wenn ein Benutzer mit einem Dialogelement interagiert, sendet Windows verschiedene Dialogelementbotschaften. Siehe Kapitel 4, um mehr über das Antworten auf Dialogelemente zu erfahren.

Ändern des Verhaltens von Fenstern in Richtung auf Dialogfenster

Bei einem Dialogfenster kann der Benutzer die Taste [*Tab*] benutzen, um zyklisch durch alle Dialogelemente des Dialogfensters zu wandern. Es ermöglicht dem Benutzer ebenso die Verwendung der Pfeiltasten, um Schaltfelder in einem Gruppenfeld auszuwählen. Um diese Tastaturschnittstelle für Fenster mit Dialogelementen zu emulieren, müssen Sie *EnableKBHandler* im Konstruktor der Fensterobjektes aufrufen.

Verwenden besonderer Dialogelemente

Jede spezielle Art von Dialogelement arbeitet etwas anders als die anderen. In diesem Abschnitt finden Sie spezielle Informationen darüber, wie die Objekte für jede der Standarddialogelemente von Windows und der speziellen Dialogelemente von ObjectWindows zu verwenden sind.

Verwenden der Listenfeld-Dialogelemente

Ein Listenfeld ist der einfachste Weg, vom Benutzer die Auswahl eines Elementes aus einer Liste zu fordern. Die Klasse *TListBox* kapselt Listenfelder ein. *TListBox* definiert Elementfunktionen für vier verschiedene Zwecke:

- Erzeugen von Listenfeldern

- Ändern der Liste von Einträgen
- Abfragen der Liste von Einträgen
  Abfragen, welcher Eintrag vom Benutzer ausgewählt wurde

Konstruieren eines Listenfeldobjektes

Einer der Konstruktoren von *TListBox* erwartet sieben Parameter: übergeordnetes Fenster, einen Ressourcen-Bezeichner, die Dimensionen *X*, *Y*, *H* und *W* des Dialogelementes sowie einen optionalen Bibliotheks-Bezeichner:

```
TListBox(TWindow *parent, int resourceId, int x, int y, int w, int h,
TLibID
libID = 0);
```

Die Standard-Dialogelementeigenschaften sind WS_CHILD, WS_VISIBLE, WS_GROUP und WS_TABSTOP

TListBox erhält die Standard-Dialogelementeigenschaften und fügt noch LBS_STANDARD hinzu. LBS_STANDARD ist eine Kombination aus LBS_NOTIFY (um Benachrichtigungsbotschaften zu empfangen), WS_VSCROLL (um eine vertikale Bildlaufleiste zu haben), LBS_SORT (um die Listeneinträge alphabetisch zu sortieren) und WS_BORDER (um einen Rand zu haben). Wenn Sie eine andere Listenfeldeigenschaft wünschen, können Sie *Attr.Style* im Konstruktor des Listenfeldobjektes oder im Konstruktor seines übergeordneten Fensters ändern. Sie könnten zum Beispiel für ein Listenfeld, das seine Einträge nicht sortiert folgenden Code verwenden:

```
listbox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
listbox->Attr.Style &= ~LBS_SORT;
```

Ändern von Listenfeldern

Nachdem Sie ein Listenfeld erzeugt haben, müssen Sie es mit Einträgen füllen, die als Strings vorliegen müssen. Später können sie Einträge anzufügen, einfügen oder entfernen. Sie können aber auch die gesamte Liste vollständig leeren. Die unten stehende Tabelle listet die Elementfunktionen für diese Aktionen auf:

Tabelle 12.2: Elementfunktionen von *TListBox* für das Ändern von Listenfeldern

Elementfunktion	Beschreibung
ClearList	Lösche jeden Eintrag
DirectoryList	Setze Dateinamen in die Liste
AddString	Füge einen Eintrag an
InsertString	Füge einen neuen Eintrag ein
DeleteString	Lösche einen Eintrag
SetSelIndex, *SetSel* oder *SetSelString*	Wähle einen Eintrag aus
SetSelStrings, *SetSelIndexes* oder *SetSelItemRange*	Wähle mehrere Einträge aus
SetTopIndex	Blättere das Listenfeld, damit der angegebene Eintrag sichtbar ist
SetTabStops	Setzte Spaltenformat für mehrspaltige Listenfelder
SetHorizontalExtent	Setze die Pixelzahl, mit der das Listenfeld horizontal blättern kann

SetColumnWidth	Setze die Breite aller Spalten bei mehrspaltigen Listenfeldern
SetCaretIndex	Setze den Index des aktuell fokussierten Eintrages
SetItemData	Setze einen DWORD-Wert, der mit dem angegebenen Index assoziiert werden soll
SetItemHeight	Setze die Höhe des Eintrages am angegebenen Index oder die Höhe aller Einträge

Abfragen der Listenfelder

Es gibt mehrere Elementfunktionen, die Sie aufrufen können, um Informationen über das Listenfeld oder dessen Einträgen zu erhalten. Die folgende Tabelle listet die Elementfunktionen für die Abfrage von Listenfeldern auf.

Tabelle 12.3: Elementfunktion von *TListBox* für die Abfrage von Listenfeldern

Elementfunktion	Beschreibung
GetCount	Zahl von Einträgen in der Liste
FindString oder *FindExactString*	Index der Position eines Strings
GetTopIndex	Index des Eintrages am Anfang des Listenfeldes
GetCaretIndex	Index des aktuell fokussierten Eintrages
GetHorizontalExtent	Pixelzahl, mit der das Listenfeld horizontal blättern kann
GetItemData	DWORD-Wert, der durch *SetItemData* gesetzt wurde
GetItemHeight	Höhe des angegebenen Eintrages in Pixeln
GetItemRect	Rechteck für die Anzeige des angegebenen Eintrages
GetSelCount	Anzahl ausgewählter Einträge
GetSelIndex oder *GetSel*	Index des ausgewählten Eintrages
GetSelString	Ausgewählter Eintrag
GetSelStrings oder *GetSelIndexes*	Ausgewählte Einträge
GetString	Eintrag zu einem bestimmten Index
GetStringLen	Länge eines bestimmten Eintrages

Antworten auf Listenfelder

Die Elementfunktionen für das Ändern und Abfragen der Listenfelder ermöglichen das Setzen von Werten und die Abfrage des Dialogelementzustandes zu jeder beliebigen Zeit. Um aber zu wissen, was der Benutzer mit dem Listenfeld zur Laufzeit macht, müssen Sie auf Benachrichtigungsbotschaften vom Dialogelement antworten.

Es gibt nur einige wenige Dinge, die der Benutzer mit einem Listenfeld unternehmen kann: blättern durch die Liste, einen Eintrag anklicken und auf einen Eintrag doppelklicken. Wenn der Benutzer eins dieser Aktionen unternimmt, sendet Windows *Listenfeld-Benachrichtigungsbotschaft* an das dem Listenfeld übergeordnete Fenster. Normalerweise definieren Sie Elementfunktionen zur Botschaftsbeantwortung im übergeordneten Fensterobjekt, um Benachrichtigungen von jedem zum übergeordneten Fenster gehörenden Dialogelement zu behandeln.

Die folgende Tabelle listet die allgemeinsten Listenfeld-Benachrichtigungen auf:

Tabelle 12.4: Listenfeld-Benachrichtigungsbotschaften

Makro der Ereignisantworttabelle	Beschreibung
EV_LBN_SELCHANGE	Ein Eintrag wurde mit einem einzelnen Mausklick ausgewählt
EV_LBN_DBLCLK	Ein Eintrag wurde mit einem einzelnen Mausklick ausgewählt
EV_LBN_SELCANCEL	Der Benutzer hat einen Eintrag deselektiert
EV_LBN_SETFOCUS	Der Benutzer hat dem Listenfeld über Klick bzw. Doppelklick auf einen Eintrag oder über die [*Tab*]-Taste den Fokus gegeben. Diese Botschaft geht der Botschaft LBN_SELCHANGE voran.
EV_LBN_KILLFOCUS	Der Benutzer hat dem Listenfeld durch das Klicken auf ein anderes Dialogelement oder durch Drücken der [*Tab*]-Taste den Fokus abgenommen.

Hier ist beispielsweise eine Elementfunktion eines übergeordneten Fensters. zur Behandlung der Botschaft LBN_SELCHANGE:

```
DEFINE_RESPONSE_TABLE1(TLBoxWindow, TFrameWindow)
  EV_LBN_SELCHANGE(ID_LISTBOX, EvListBoxSelChange),
END_RESPONSE_TABLE;

void TLBoxWindow::EvListBoxSelChange()
{
  int index = ListBox->GetSelIndex();
  if (ListBox->GetStringLen(index)  10) {
    char string[10];
    ListBox->GetSelString(string, sizeof(string));
    MessageBox(string, "You selected:", MB_OK);
  }
}
```

Verwenden von statischen Dialogelementen

Statische Dialogelemente sind normalerweise unveränderliche Einheiten von Text oder einfache Grafik. Der Benutzer interagiert nicht mit statischen Dialogelementen, auch wenn Ihre Anwendung den Text von statischen Dialogelementen ändern kann.

Siehe OWLAPI\STATIC für ein Beispiel zu statischen Dialogelementen.

Konstruieren von statischen Dialogelementobjekten

Da der Benutzer niemals direkt mit einem statischen Dialogelement interagiert, empfängt die Anwendung keine Dialogelement-Benachrichtigungsbotschaften von statischen Dialogelementen. Deshalb können Sie die meisten statischen Dialogelemente mit -1 als Dialogelement-ID konstruieren. Wenn Sie allerdings *TWindow::SendDlgItemMessage* verwenden

wollen, um das statische Dialogelement zu manipulieren, brauchen Sie eine eindeutige ID.

Der Konstruktor von *TStatic* ist wie folgt deklariert:

```
TStatic(TWindow *parent, int resourceId, const char far *title, int x,
int y,
int w, int h, UINT textLen, TLibID libID = 0);
```

Er erwartet die normalen sieben Parameter (übergeordnetes Fenster, einer Ressource-ID, die Dimensionen des Dialogelementes *X*, *Y*, *H* und *W* und eine optionale Bibliotheks-ID) sowie zwei für statische Dialogelemente spezielle Parameter: ein Text-String, den das statische Dialogelement anzeigt und dessen maximale Länge (einschließlich der terminierenden NULL). Ein typischer Aufruf zum Konstruieren eines statischen Dialogelementes sieht wir folgt aus:

```
new TStatic (this, -1, "Sample &Text", 170, 20, 200, 24, 0);
```

Wenn Sie sich die Möglichkeit offen halten wollen, den Text eines statischen Dialogelementes zu ändern, benötigen Sie ein Datenelement im übergeordneten Fensterobjekt, um die Elementfunktion des statischen Dialogelementobjektes aufrufen zu können. Wenn der Text sich nicht ändern wird benötigen Sie kein Datenelement

Die Standard-Dialogelementeigenschaften sind WS_CHILD, WS_VISIBLE, WS_GROUP und WS_TABSTOP

TListBox erhält die Standard-Dialogelementeigenschaften, fügt SS_LEFT hinzu (um den Text linksbündig darzustellen) und löscht die Eigenschaft WS_TABSTOP (um zu verhindern, daß der Benutzer das Dialogelement über die [*Tab*]-Taste anwählt). Um die Eigenschaften zu ändern, müssen Sie *Attr.Style* im Konstruktor des statischen Dialogelementobjektes modifizieren. Wenn Sie zum Beispiel den Text des Dialogelementes zentrieren wollen, schreiben Sie:

```
Attr.Style = (Attr.Style & ~SS_LEFT) | SS_CENTER;
```

Eine Option für statische Dialogelemente ist das Unterstreichen von einem oder mehreren Zeichen im Text-String, um Mnemos für nahe gelegene Dialogelemente anzuzeigen. Fügen Sie ein kaufmännisches Und `&` im String unmittelbar vor dem zu unterstreichenden Zeichen ein. Um zum Beispiel das `T` in `Text` zu unterstreichen, können Sie `&Text` verwenden. Wenn Sie ein kaufmännisches Und innerhalb des String benutzen wollen, können Sie die statische Eigenschaft SS_NOPREFIX verwenden.

Ändern statischer Dialogelemente

TStatic besitzt zwei Elementfunktionen für die Veränderung des Textes eines statischen Dialogelementes. *SetText* setzt den Text auf den übergebenen String. *Clear* löscht den Text. Sie können den Text von statischen Dialogelementen nicht ändern, wenn sie mit der Eigenschaft SS_SIMPLE erzeugt wurden.

Abfragen statischer Dialogelemente

TStatic::GetTextLen gibt die Textlänge des statischen Dialogelementes wieder. Verwenden Sie *TStatic::GetText*, um den Text selbst zu erhalten.

Verwenden von Schaltern als Dialogelemente

Schalter (manchmal auch Aktionsschalter oder Kommandoschalter genannt) führen jedesmal dann eine Aufgabe aus, wenn der Schalter gedrückt wurde. Es gibt zwei Arten von Schaltern: Standardschalter und nicht-Standardschalter. Ein Standardschalter, unterschieden durch die Schaltereigenschaft BS_DEFPUSHBUTTON, hat einen dicken Rand, der die vorgegebene Antwort des Benutzers anzeigt. Nicht-Standardschalter besitzen die Schaltereigenschaft BS_PUSHBUTTON.

Siehe \OWLAPI\BWCC für Beispiele zu Schaltern als Dialogelemente.

Konstruieren von Schaltern

Einer der Konstruktoren von *TButton* erwartet die sieben Parameter, die allgemein in einem Konstruktor eines Dialogelementes zu finden sind (ein übergeordnetes Fenster, einen Ressourcen-Bezeichner, die Dimensionen *x*, *y*, *h* und *w* des Dialogelementes und optional einen Bibliotheks-Bezeichner). Zusätzlich dazu erwartet er einen Text-String, der die Beschriftung des Schalters angibt, und ein BOOL-Flag, das angibt, ob der Schalter ein Standardschalter sein soll. Hier ist die Deklaration des Konstruktors:

```
TButton(TWindow *parent, int resourceId, const char far *text, int X, int Y,
        int W, int H, BOOL isDefault = FALSE, TLibID libID = 0);
```

Ein typischer Schalter würde wie folgt konstruiert:

```
btn = new TButton(this, ID_BUTTON, "DO_IT!", 38, 48, 316, 24, TRUE);
```

Antworten auf Schalter

Wenn der Benutzer auf einen Schalter klickt, empfängt das zum Schalter übergeordnete Fenster eine Benachrichtigungsbotschaft. Wenn das übergeordnete Fensterobjekt die Botschaft abfängt, kann es auf diese Ereignisse antworten, indem es ein Dialogfenster aufwirft, eine Datei sichert usw..

Um Schalterbotschaften, abzufangen und zu beantworten, müssen Sie eine Elementfunktion zur Kommandobeantwortung für die Schalter definieren. Das folgende Beispiel behandelt die Antwort auf einen Benutzer, der den Schalter mit der ID ID_SCHALTER anklickt.

```
DEFINE_RESPONSE_TABLE1(TTestWindow, TFrameWindow)
  EV_COMMAND(ID_SCHALTER, HandleButtonMsg),
END_RESPONSE_TABLE;

void TTestWindow::HandleButtonMsg()
{
  // Schalter wurde gedrückt
}
```

Verwenden von Markierungsfeldern und Schaltfeldern als Dialogelemente

Ein *Markierungsfeld* konfrontiert den Benutzer im allgemeinen mit zwei Wahlmöglichkeiten. Der Benutzer kann das Dialogelement markieren, die Markierung aufheben oder das Element lassen wie es ist. In einer Gruppe von Markierungsfeldern kann eine beliebige Auswahl oder auch alle Felder markiert werden. Sie könnten zum Beispiel ein Markierungsfeld dazu verwenden, die Ausgabe von Tönen in Ihrer Anwendung ein- oder auszuschalten.

Schaltfelder andererseits dienen für die Auswahl von einer oder mehrerer wechselseitig ausgeschlossener Optionen. Sie könnten zum Beispiel Schaltfelder dazu verwenden, zwischen einer Reihe von Tönen in Ihrer Anwendung auszuwählen.

TCheckBox ist von *TButton* abgeleitet und repräsentiert Markierungsfelder. Da Schaltfelder einige Verhalten mit Markierungsfelder teilen, ist *TRadioButton* von *TCheckBox* abgeleitet. Markierungsfelder und Schaltfelder werden manchmal zusammenfassend als *Auswahlfelder* bezeichnet.

Ein wichtiger Punkt im Zusammenhang mit Auswahlfeldern ist ihr Zustand. Während ein Auswahlfeld auf dem Bildschirm angezeigt wird, ist es entweder markiert oder nicht markiert. Wenn der Benutzer ein Auswahlfeld anklickt, ruft er damit ein Ereignis hervor, das eine Windows-Botschaft generiert. Wie bei anderen Dialogelementen fängt das dem Auswahlfeld übergeordnete Fenster normalerweise die Botschaften ab und reagiert auf diese.

Siehe OWLAPI\BUTTON für Beispiele zu Markierungsfeldern und Schaltfeldern als Dialogelemente.

Konstruieren von Markierungsfeldern und Schaltfeldern

TCheckBox und *TRadioButton* besitzen jeweils einen Konstruktor, der die sieben Parameter erwartet, die allgemein in einem Konstruktor eines Dialogelementes zu finden sind (ein übergeordnetes Fenster, einen Ressourcen-Bezeichner, die Dimensionen *x*, *y*, *h* und *w* des Dialogelementes und optional einen Bibliotheks-Bezeichner). Zusätzlich dazu erwarteten die Konstruktoren einen Test-String und einen Zeiger auf ein Gruppenfeldobjekt, das die Auswahlfelder gruppiert. Wenn der Zeiger auf das Gruppenfeldobjekt Null enthält, ist das Auswahlfeld nicht teil eines Gruppenfeldes. Hier sind jeweils einer der Konstruktoren:

```
TCheckBox(TWindow *parent, int resourceId, const char far *title, int x,
          int y, int w, int h, TGroupBox *group, TLibID libID = 0);

TRadioButton(TWindow *parent, int resourceId, const char far *title, int x,
             int y, int w, int h, TGroupBox *group, TLibID libID = 0);
```

Das unten stehende Listing zeigt einige typische Aufrufe von Konstruktoren für Auswahlfelder.

```
CheckBox = new TCheckBox(this, ID_CHECKBOX, "Check Box Text", 158, 12,
                         150, 26, 0);
GroupBox = new TGroupBox(this, ID_GROUPBOX, "Group Box", 158, 102,
                         176, 108);
RButton1 = new TRadioButton(this, ID_RBUTTON1, "Radio Button 1",
                            174, 128, 138, 24, GroupBox);
RButton2 = new TRadioButton(this, ID_RBUTTON2, "Radio Button 2",
                            174, 162, 138, 24, GroupBox);
```

Markierungsfelder haben standardmäßig die Eigenschaft BS_AUTOCHECKBOX, die bedeutet, daß Windows einen Klick auf das Markierungsfeld mit seinem Umschalten beantwortet. Ohne BS_AUTOCHECKBOX müßten Sie den Zustand des Markierungsfeldes manuell einstellen. Schaltfelder haben standardmäßig die Eigenschaft BS_AUTORADIOBUTTON, die bedeutet, daß Windows einen Klick auf das Schaltfeld damit beantwortet, indem es das Schaltfeld markiert und die Markierung aller anderen Schaltfelder in der Gruppe aufhebt. Ohne BS_AUTORADIOBUTTON müßten Sie die Benachrichtigungsbotschaften des Schaltfeldes abfangen und selbst behandeln.

Ändern von Auswahlfeldern

Das Ändern des Zustandes eines Auswahlfeldes (durch Markieren und Aufheben der Markierung) scheint eher eine Aufgabe für den Benutzer und nicht für Ihre Anwendung zu sein. Aber in einigen Fällen benötigt Ihre Anwendung die Kontrolle über den Zustand des Auswahlfeldes. Wenn der Benutzer zum Beispiel eine Textdatei öffnet, könnten Sie automatisch ein Markierungsfeld mit der Beschriftung "Save as ANSI text." markieren. *TCheckBox* definiert mehrere Elementfunktionen, um den Zustand eines Markierungsfeldes zu ändern:

Tabelle 12.5: Elementfunktionen von *TCheckBox* für die Änderung von Auswahlfeldern

Elementfunktion	Beschreibung
Check oder *SetCheck(BF_CHECKED)*	Markieren
Uncheck oder *SetCheck(BF_UNCHECKED)*	Markierung aufheben
Toggle	Umschalten
SetState	Hervorheben
SetStyle	Ändern der Schaltereigenschaft

Wenn Sie diese Elementfunktionen bei Schaltfeldern verwenden, stellt ObjectWindows sicher, daß nur ein Schaltfeld pro Gruppe markiert ist, solange die Schalter einer Gruppe zugewiesen sind.

Abfragen von Auswahlfeldern

Die Abfrage eines Auswahlfeldes ist ein Weg, dessen Zustand herauszufinden und darauf zu antworten. Schaltfelder besitzen zwei Zustände: markiert (BF_CHECKED) und nicht markiert (BF_UNCHECKED). Markierungsfelder besitzten einen optionalen dritten Zustand: grau dargestellt (BF_GRAYED). Die unten stehende Tabelle faßt die Abfrage-Elementfunktionen für ein Auswahlfeld zusammen.

Tabelle 12.6: Elementfunktionen von *TCheckBox* für die Abfrage von Auswahlfeldern

Elementfunktion	Beschreibung
GetCheck	Zurückgeben des Markierungszustandes
GetState	Zurückgeben des Markierungs-, Hervorhebungs- oder Fokuszustandes

Verwenden von Gruppenfeldern

In seiner einfachsten Form ist ein Gruppenfeld ein beschriftetes statisches Rechteck, das andere Dialogelemente sichtbar gruppiert.

Konstruieren von Gruppenfeldern

TGroupBox besitzt einen Konstruktor, der die sieben Parameter, die allgemein in einem Konstruktor eines Dialogelementes zu finden sind (ein übergeordnetes Fenster, einen Ressourcen-Bezeichner, die Dimensionen x, y, h und w des Dialogelementes und optional einen Bibliotheks-Bezeichner), erwartet. Zusätzlich dazu erwartet er einen Text-String, um die Gruppe zu beschriften:

```
TGroupBox(TWindow *parent, int resourceId, const char far *text, int X,
          int Y, int W, int H, TLibID libID = 0);
```

Gruppieren von Dialogelementen

Während ein Gruppenfeld eine Gruppe anderer Dialogelemente sichtbar assoziiert, kann es auch eine Gruppe von Auswahlfeldern lokal assoziieren (Markierungsfelder und Schaltfelder). Diese logische Gruppierung führt das automatische Aufheben von Markierungen aus.

Um ein Auswahlfeld einem Gruppenfeld hinzuzufügen, müssen Sie einen Zeiger auf das Gruppenfeldobjekt im Konstruktor des Auswahlfeldes übergeben.

Antworten auf Gruppenfelder

Wenn ein Ereignis eintritt, das die Auswahlen in einem Gruppenfeld ändern sollte (z.B. wenn ein Benutzer auf einen Schalter klickt oder die Anwendung *Check* aufruft), sendet Windows eine Benachrichtigungsbotschaft an das zum Gruppenfeld übergeordnete Fenster. Das übergeordnete Fenster kann die Botschaft für das Gruppenfeld als Ganzes abfangen, anstatt auf die individuellen Auswahlfelder im Gruppenfeld zu antworten. Um herauszufinden, welches Dialogelement in der Gruppe betroffen war, können Sie den aktuellen Zustand von jedem Dialogelement lesen.

Verwenden von Bildlaufleisten

Bildlaufleisten sind der primäre Mechanismus, um das View des Benutzers einer Anwendung, ein Listenfeld oder ein Kombinationsfenster zu ändern. Es kann aber auch sein, daß Sie eine separate Bildlaufleiste für eine spezielle Aufgabe benötigen. Diese Aufgabe kann beispielsweise die Kontrolle der Temperatur bei einem Thermostat oder die Farbe in einem Zeichenprogramm sein. Verwenden Sie *TScrollBar*-Objekte, wenn Sie eine separate, für Ihre Zwecke verwendbare Bildlaufleiste benötigen.

Siehe OWLAPI\SCROLLER für Beispiele zu Bildlaufleisten als Dialogelemente.

Konstruieren von Bildlaufleisten

TScrollBar besitzt einen Konstruktor, der die sieben Parameter, die allgemein in einem Konstruktor eines Dialogelementes zu finden sind (ein übergeordnetes Fenster, einen Ressourcen-Bezeichner, die Dimensionen *x*, *y*, *h* und *w* des Dialogelementes und optional einen Bibliotheks-Bezeichner), erwartet. Zusätzlich dazu erwartet er ein BOOL-Flag als Parameter, das angibt, ob die Bildlaufleiste horizontal ist. Hier ist die Konstruktordeklaration von *TScrollBar*:

```
TScrollBar(TWindow *parent, int resourceId, int x, int y, int w, int h,
           BOOL isHScrollBar, TLibID libID = 0);
```

Wenn Sie eine Breite von Null für eine vertikale Bildlaufleiste oder eine Höhe von Null für eine horizontale Bildlaufleiste angeben, teilt ihr Windows eine Standardbreite oder -höhe zu. Der folgende Code erzeugt eine horizontale Bildlaufleiste mit Standardhöhe:

```
new TScrollBar(this, ID_THERMOMETER, 100, 150, 180, 0, TRUE);
```

Der Konstruktor von *TScrollBar* konstruiert Bildlaufleisten mit der Eigenschaft SBS_HORZ für horizontale Bildlaufleisten und SBS_VERT für vertikale Bildlaufleisten. Sie können zusätzliche Eigenschaften angeben wie zum Beispiel SBS_TOPALIGN, indem Sie das Element *Attr.Style* des Bildlaufleistenobjektes ändern.

Kontrollieren des Bildlaufleistenbereiches

Ein Attribut einer Bildlaufleiste ist ihr *Bereich*. Der Bereich einer Bildlaufleiste ist die Menge aller möglichen *Positionszeiger*-Stellungen. Der Positionszeiger ist das Gleitreglerfeld der Bildlaufleiste, welches der Benutzer zieht oder blättert. Jede Position ist mit einer ganzen Zahl assoziiert. Das übergeordnete Fenster benutzt diese Ganzzahl, die *Position*, um die Bildlaufleiste abzufragen und zu setzen. Standardmäßig ist der Bereich eines Bildlaufleistenobjektes von 1 bis 100.

Die minimale Position des Positionszeigers (am oberen Ende der vertikalen Bildlaufleiste und am linken Ende der horizontalen Bildlaufleiste) korrespondiert zum Wert 1. Entsprechend korrespondiert die maximale Position des Positionszeigers zum Wert 100. Verwenden Sie *SetRange*, um den Bereich umzusetzen.

Kontrollieren des Blätterumfangs

Eine Bildlaufleiste besitzt zwei wichtige Attribute: ihre Zeilengröße und ihre Seitengröße. Die Zeilengröße ist mit 1 initialisiert und gibt die Weite in *Bereich*-Einheiten an, um die sich der Positionszeiger bewegt, wenn der Benutzer auf die Bildlaufleistenpfeile klickt. Die Seitengröße ist mit 10 initialisiert und gibt die Weite ebenfalls in *Bereich*-Einheiten an, um die sich der Positionszeiger bewegt, wenn der Benutzer auf das Feld zum Blät-

tern klickt. Über die Datenelemente *LineMagnitude* und *PageMagnitude* von *TScrollBar* können Sie diese Werte ändern.

Abfragen von Bildlaufleisten

TScrollBar besitzt zwei Elementfunktionen für die Abfrage von Bildlaufleisten:

- *GetRange* erhält die Ober- und Untergrenze des Bereiches
- GetPosition erhält die aktuelle Stellung des Positionszeigers

Ändern von Bildlaufleisten

Das Ändern von Bildlaufleisten ist meistens die Aufgabe des Benutzers. Aber auch Ihre Anwendung kann die Bildlaufleiste direkt verändern:

- *SetRange* setzt den Bereich zum Blättern
- SetPosition setzt die Stellung des Positionszeigers
- DeltaPos verändert die Stellung des Positionszeigers

Antworten auf Botschaften von Bildlaufleisten

Wenn der Benutzer den Positionszeiger der Bildlaufleiste bewegt oder auf die Pfeile zum Blättern klickt, sendet Windows Benachrichtigungsbotschaft für Bildlaufleisten an das übergeordnete Fenster. Wenn Ihr Fenster auf die Blätter-Ereignisse antworten soll, muß des auf die Benachrichtigungsbotschaften reagieren.

Benachrichtigungsbotschaften für Bildlaufleisten unterscheiden sich gering von anderen Dialogelement-Benachrichtigungsbotschaften. Sie basieren auf den Botschaften WM_HSCROLL und WM_VSCROLL, nicht aber auf Kommandobotschaften WM_COMMAND. Deshalb müssen Sie die Funktionen *EvHScroll* oder *EvVScroll* zur Beantwortung von Ereignissen definieren, damit Sie auf die Benachrichtigungsbotschaften für Bildlaufleisten antworten können. Die Auswahl der beiden Funktionen hängt davon ab, ob die Bildlaufleiste horizontal oder vertikal ist:

```
class TTestWindow : public TFrameWindow {
  public:
    TTestWindow(TWindow* parent, const char* title);
    virtual void SetupWindow();

    void EvHScroll(UINT code, UINT pos, HWND wnd);

  DECLARE_RESPONSE_TABLE(TTestWindow);
};

DEFINE_RESPONSE_TABLE1(TTestWindow, TFrameWindow)
  EV_WM_HSCROLL,
END_RESPONSE_TABLE;
```

In der Regel reagieren Sie auf alle Benachrichtigungsbotschaften für Bildlaufleisten, indem Sie die aktuelle Stellung des Positionszeigers erfragen und eine passende Aktion auslösen. In diesem Fall können Sie den Benachrichtigungscode ignorieren:

```
void TTestWindow::EvHScroll(UINT code, UINT pos, HWND wnd)
{
  TFrameWindow::EvHScroll(); // Führe Standardbearbeitung von WM_HSCROLL aus
  int newPos = ScrollBar->GetPosition();
  // Bearbeitung im Zusammenhang mit newPos
}
```

Vermeiden von Verfolgungsbotschaften für Positionszeiger

Eventuell wollen Sie nicht auf Benachrichtigungsbotschaften der Bildlaufleiste antworten, solange der Benutzer den Positionszeiger der Bildlaufleiste zieht. Der Benutzer zieht nämlich den Positionszeiger oft sehr schnell, so daß viele Benachrichtigungsbotschaften generiert werden. Es ist effizienter, zu warten bis der Benutzer die Bewegung des Positionszeiger beendet hat und dann erst zu antworten. Dazu müssen Sie die Benachrichtigungsbotschaften mit dem Code SB_THUMBTRACK ausblenden.

Spezialisieren des Bildlaufleistenverhaltens

Eventuell wollen Sie, daß das Bildlaufleistenobjekt selbst auf seine Benachrichtigungsbotschaften antwortet, wodurch ein spezielles Verhalten in das Bildlaufleistenobjekt eingesetzt werden muß. *TWindow* besitzt eine eingebaute Unterstützung, um Benachrichtigungsbotschaften für Bildlaufleisten zur Bildlaufleiste weiterzureichen. *TWindow::EvHScroll* oder *TWindow::EvVScroll* führen die passende Elementfunktion von *TScrollBar*, basierend auf dem Benachrichtigungscode aus. Ein Beispiel:

```
class TSpecializedScrollBar : public TScrollBar {
public:
  virtual void SBTop();
  };

void TSpecializedScrollBar::SBTop() {
  TScrollBar::SBTop();
  sndPlaySound("AT-TOP.WAV", SND_ASYNC); // spiele einen Ton
  }
```

Stellen Sie sicher, daß Sie zuerst die Elementfunktionen der Basis aufrufen. Diese Funktionen aktualisieren die Bildlaufleiste für ihre neue Position in korrekter Weise.

Die folgende Tabelle assoziiert Benachrichtigungsbotschaften mit der entsprechenden Elementfunktion von *TScrollBar*:

Tabelle 12.7: Benachrichtigungscodes und Elementfunktionen von *TScrollBar*

Benachrichtigungsbotschaft	Elementfunktion von *TScrollBar*
B_LINEUP	*SBLine*
SB_LINEDOWN	*SBLine*
SB_PAGEUP	*SBPage*
SB_PAGEDOWN	*SBPage*
SB_THUMBPOSITION	*SBThumbPosition*

SB_THUMBTRACK	*SBThumbTrack*
SB_TOP	*SBTop*
SB_BOTTOM	*SBBottom*

Verwenden von Gleitreglern und Meßanzeigen

Gleitregler sind spezialisierte Bildlaufleisten Die Klasse *TSlider* ist von *TScrollBar* abgeleitet. Gleitregler werden für Positionsinformation benutzt, die nicht zum Blättern gedacht ist. Die zwei von *TSlider* abgeleiteten Klassen *THSlider* und *TVSlider* implementieren vertikale und horizontale Gleitreglerversionen.

Meßanzeigen sind Dialogelemente, die die Dauer oder andere Information über einen fortschreitenden Prozeß anzeigen. Die Klasse *TGauge* implementiert Meßanzeigen und ist von der Klasse *TControl* abgeleitet. Ein Parameter für den Konstruktor bestimmt, ob Sie eine vertikale oder horizontale Meßanzeige erhalten. Horizontale Meßanzeigen dienen üblicherweise der Anzeige von Prozeßinformationen und vertikale Meßanzeigen dienen üblicherweise der Anzeige von analoger Information.

Siehe OWLAPI\SLIDER für Beispiele zu Gleitreglern und Meßanzeigen als Dialogelemente.

Verwenden von Editierfeldern

Editierfelder sind interaktive statische Dialogelemente. Sie sind rechteckige Felder, die mit Text gefüllt werden können und durch den Benutzer oder die Anwendung verändert oder geleert werden. Editierfelder sind sehr nützlich als Felder zur Dateneingabe. Sie unterstützen die folgenden Operationen:

- Texteingabe durch den Benutzer
- Dynamische Anzeige von Text (durch die Anwendung)
- Ausscheiden, Kopieren und Einfügen in Zusammenarbeit mit der Zwischenablage
- Mehrzeiliges Editieren (Nützlich bei Texteditoren)

Siehe OWLAPI\VALIDATE für Beispiele zu Editierfledern als Dialogelemente.

Konstruieren von Editierfeldern

Einer der Konstruktoren von *TEdit* erwartet Parameter für einen initialen Text-String, die maximale Länge (einschließlich der terminierenden NULL) und ein BOOL-Flag, das angibt, ob es sich um ein mehrzeiliges Editierfeld handelt (zusätzlich zum übergeordneten Fenster, dem Ressourcen-Bezeichner und den Plazierungskoordinaten) . Dieser *TEdit*-Konstruktor ist wie folgt deklariert:

```
TEdit(TWindow *parent, int resourceId, const char far *text, int x,
      int y, int w, int h, UINT textLen, BOOL multiline = FALSE,
      TLibID libID = 0);
```

Standardmäßig besitzt das Editierfeld die Eigenschaften ES_LEFT (für linksbündigen Text), ES_AUTOHSCROLL (für automatisches horizontales Blättern) und WS_BORDER (für einen sichtbaren Rand, der das Editierfeld umgibt). Mehrzeilige Editierfelder erhalten die zusätzlichen Eigenschaften ES_MULTILINE (spezifiziert ein mehrzeiliges Editierfeld), ES_AUTOVSCROLL (für automatisches vertikales Blättern), WS_VSCROLL (vertikale Bildlaufleiste) und WS_HSCROLL (horizontale Bildlaufleiste).

Die folgenden Code-Zeilen stellen typische Konstruktoraufrufe für Editierfelder dar, der eine für ein einzeiliges Editierfeld und der andere für ein mehrzeiliges:

```
Edit1 = new TEdit(this, ID_EDIT1, "Default Text", 20, 50, 150, 30,
                  MAX_TEXTLEN, FALSE);
Edit2 = new TEdit(this, ID_EDIT2, "", 260, 50, 150, 30, MAX_TEXTLEN,
                  FALSE);
```

Verwenden der Zwischenablage und des Bearbeiten-Menüs

Sie können Text zwischen einem Editierfeldobjekt und der Zwischenablage von Windows über Elementfunktionen von *TEdit* direkt transferieren.

Sie können auch dem Benutzer den Zugriff auf diese Elementfunktionen erlauben, indem Sie für Ihr Fenster ein *Bearbeiten*-Menü einbauen.

Editierfeldobjekte haben eingebaute Antworten für Menüeinträge wie *Bearbeiten | Kopieren* und *Bearbeiten | Rückgängig*. *TEdit* hat Elementfunktionen zur Beantwortung von Kommandos wie beispielsweise *CmEditCopy* und *CmEditUndo*. Diese Funktionen werden von ObjectWindows aufgerufen, wenn Benutzer Einträge aus dem *Bearbeiten*-Menü des übergeordneten Fensters auswählen.

Die unten stehende Tabelle zeigt die Elementfunktionen für die Zwischenablage und für das Editieren sowie die Menükommandos, die die Ausführung dieser Funktionen hervorrufen.

Tabelle 12.8: Elementfunktionen von *TEdit* und *Bearbeiten*-Menükommandos

Elementfunktion	Menükommando	Operation
Copy	CM_EDITCOPY	Kopiere den Text in die Zwischenablage
Cut	CM_EDITCUT	Schneide den Text für die Zwischenablage aus
Undo	CM_EDITUNDO	Mache die letzte Bearbeitung rückgängig
Paste	CM_EDITPASTE	Füge den Text aus der Zwischenablage ein
DeleteSelection	CM_EDITDELETE	Lösche den markierten Text
Clear	CM_EDITCLEAR	Leere das gesamte Editierfeld

Um ein Bearbeitungsmenü einem Fenster, das Editierfeldobjekte enthält, hinzuzufügen, müssen Sie eine Menü-Ressource für das Fenster mit den oben aufgelisteten Menükommandos definieren. Sie müssen keine neuen Elementfunktionen schreiben.

Abfragen von Editierfeldern

Oftmals wollen Sie ein Editierfeld abfragen, um ein Eingabe für spätere Zwecke zu speichern. *TEdit* besitzt eine Reihe von Elementfunktionen für eine solche Abfrage. Viele der Elementfunktionen eines Editierfeldes für die Abfrage oder Veränderung erfordern die Angabe einer Zeilennummer oder einer Zeichenposition in einer Zeile bzw. geben diese Werte zurück. Für beides startet der Index bei Null. Mit anderen Worten ist die erste Zeile die Zeile Null und das erste Zeichen einer Zeile das Zeichen Null. Die unten stehende Tabelle faßt die Elementfunktionen zur Abfrage von *TEdit* zusammen.

Tabelle 12.9: Elementfunktionen von *TEdit* für die Abfrage von Editierfeldern

Beschreibung	Elementfunktion
IsModified	Überprüfe, ob der Text verändert wurde
GetText	Rufe den gesamten Text ab
GetLine	Rufe eine Zeile ab
GetNumLines	Gibt die Zahl von Zeilen zurück
GetLineLength	Gib die Länge der gegebenen Zeile zurück
GetSelection	Gib den Index des ausgewählten Textes zurück
GetSubText	Gib einen Bereich von Zeichen zurück
GetLineIndex	Zähle die Zeichen vor einer Zeile
GetLineFromPos	Finde die Zeile, die einen Index enthält
GetRect	Gib das formatierende Rechteck zurück
GetHandle	Gib das Speicher-Handle zurück
GetFirstVisibleLine	Gib den Index der ersten sichtbaren Zeile zurück
GetPasswordChar	Gib die in Paßworten verwendeten Zeichen zurück
GetWordBreakProc	Gib die Prozedur zum Wortumbruch zurück
CanUndo	Überprüfe, ob die Bearbeitung rückgängig gemacht werden kann

Ein Text, der in einem mehrzeiligen Editierfeld die Länge einer Zeile überragt, enthält zwei besondere Zeichen für jeden Zeilenumbruch: einen Zeilenrücklauf (’\r’) und einen Zeilenvorschub (’\n’). Elementfunktionen von *TEdit* behalten die Formatierung des Texte bei, wenn Sie Text aus einem mehrzeiligen Editierfeld zurückgeben. Wenn Sie den Text in ein Editierfeld wieder einfügen, von der Zwischenablage aus einfügen, ihn in eine Datei Schreiben oder ihn auf dem Drucker ausdrucken, erscheinen die Zeilenumbrüche auch dort in der Form, wie sie im Editierfeld vorhanden waren. Berücksichtigen Sie die beiden extra Zeichen für einen Zeilenumbruch, wenn Sie die Abfrage-Elementfunktionen verwenden, die ein angegebene Zahl von Zeichen ermitteln.

Ändern von Editierfeldern

Viele Verwendungen von Editierfeldern erfordern es, daß Ihre Anwendung einen Text explizit ersetzt, einfügt, löscht oder auswählt. *TEdit* unterstützt solche Operationen und ermöglicht das Blättern eines Editierfeldes.

Tabelle 12.10: Elementfunktionen von *TEdit* für das Ändern von Editierfeldern

Beschreibung	Elementfunktion
Clear	Lösche gesamten Text
DeleteSelection	Lösche ausgewählten Text
DeleteSubText	Lösche Bereich von Zeichen
DeleteLine	Lösche eine Textzeile
Insert	Füge Text ein
Paste	Füge Text von der Zwischenablage ein
SetText	Ersetze gesamten Text
SetSelection	Wähle einen Textbereich aus
Scroll	Blättere den Text
ClearModify	Lösche das *Modified*-Flag
Search	Suche nach Text
SetRect oder *SetRectNP*	Setze das Formatierrechteck
FormatLines	Schaltet weiche Zeilenumbrüche an/aus
SetTabStops	Setze Spaltenformat
SetHandle	Setze Handle des lokalen Speichers
SetPasswordChar	Setze Paßwortzeichen
SetReadOnly	Erlaube nur Lesezugriff auf das Editierfeld
SetWordBreakProc	Setze Prozedur für den Wortumbruch
EmptyUndoBuffer	Leere den Puffer für das Zurücknehmen von Änderungen

Verwenden von Kombinationsfenstern

Ein Kombinationsfenster-Dialogelement ist eine Kombination von zwei anderen Dialogelementen: ein Listenfeld und ein Editierfeld. Es dient dem gleichen Zweck wie ein Listenfeld – es ermöglicht dem Benutzer die Auswahl eines Texteintrages von einer verschiebbaren Liste von Texteinträgen, indem auf sie mit der Maus geklickt wird. Das Editierfeld das an die Spitze des Listenfeldes gesetzt ist, bietet einen anderen Auswahlmechanismus. Die Benutzer können darüber den Text des gewünschten Eintrages eingeben. Wenn der Listenfeldbereich des Kombinationsfensters angezeigt wird, wird der gewünschte Eintrag automatisch ausgewählt. *TComboBox* ist von *TListBox* abgeleitet und erbt deren Elementfunktionen für das Ändern, Abfragen und Auswählen von Listeneinträgen. Zusätzlich bietet *TComboBox* Elementfunktionen für die Manipulation eines Listenteiles des Kombinationsfensters, der bei einigen Typen von Kombinationsfenstern, bei Bedarf *aufklappen* kann.

Siehe OWLAPI\OWLCMD für Beispiele zu Kombinationsfenstern als Dialogelemente.

Verschiedene Formen von Kombinationsfenstern

Es gibt drei Typen von Kombinationsfenstern: einfache, aufklappbare und nicht editierbare. Alle Kombinationsfenster zeigen zu jeder Zeit ihre Editierbereiche, aber einige können ihre Listenfeldbereiche anzeigen und verbergen. Die unten stehende Abbildung zeigt die Erscheinungsform der drei Kombinationsfenstertypen wie auf die eines Listenfeldes. Die un-

ten stehende Tabelle faßt die Eigenschaften von jedem Kombinationsfenstertyp zusammen.

Tabelle 12.11: Zusammenfassung von Kombinationsfenstereigenschaften

Eigenschaft	Kann Liste verbergen	Text muß mit Liste übereinstimmen
einfach	Nein	Nein
aufklappbar	Ja	Nein
nicht editierbar	Ja	Ja

Aus der Perspektive eines Benutzers unterscheiden sich die verschiedenen Eigenschaften eines Kombinationsfensters wie folgt:

- Ein einfaches Kombinationsfenster kann seinen Listenfeldbereich nicht verbergen. Sein Editierbereich verhält sich genauso wie ein Editierfeld. Der Benutzer kann Text eingeben und bearbeiten und der Text muß nicht mit einem der Einträge in der Liste übereinstimmen. Wenn er übereinstimmt, wird der korrespondierende Listeneintrag ausgewählt.
- Ein aufklappbares Kombinationsfenster verhält sich wie ein einfaches Kombinationsfenster mit einer Ausnahme. In seinem Anfangszustand wird sein Listenbereich nicht angezeigt. Er erscheint, wenn der Benutzer auf das Symbol rechts vom Editierbereich klickt. Wenn sie nicht verwendet werden, nehmen sie wesentlich weniger Raum auf als ein einfaches Kombinationsfenster oder ein Listenfeld.
- Der Listenbereich eines nicht editierbaren Kombinationsfensters verhält sich wie der Listenbereich eines aufklappbaren Kombinationsfensters – er erscheint nur, wenn benötigt. Die beiden Kombinationsfenstertypen unterscheiden sich im Verhalten ihrer Editierbereiche. Wo die Editierbereiche bei aufklappbaren Kombinationsfenstern wie reguläre Editierfelder arbeiten, sind die Editierbereiche von nicht editierbaren Kombinationsfenstern dahingehend beschränkt, daß sie nur den Text von einer der Listeneinträge anzeigen können. Wenn der Editiertext mit dem Eintragstext übereinstimmt, kann kein weiteres Zeichen mehr eingegeben werden.

Auswählen eines Kombinationsfenstertyps

Nicht editierbare Kombinationsfenster sind in Fällen sinnvoll, wo keine andere Auswahl außer denen im Listenfeld angezeigten akzeptabel ist. Wenn Sie zum Beispiel einen Drucker auswählen wollen, können Sie nur einen Drucker, der in Ihrem System zur Verfügung steht, wählen. Auf der anderen Seite können aufklappbare Kombinationsfenster andere Einträge als die in der Liste akzeptieren. Eine Verwendung für aufklappbare Kombinationsfenster ist die Auswahl von Dateien zum Öffnen oder Speichern. Der Benutzer kann entweder die Verzeichnisse durchsuchen, um die passende Datei in der Liste zu finden, oder den vollständigen Pfadnamen und Dateinamen im Editierbereich eingeben. Diese Eingabe ist unabhängig davon, ob der Dateiname im Listenbereich erscheint.

Konstruieren von Kombinationsfenstern

TComboBox besitzt einen Konstruktor, der die sieben Parameter, die allgemein in einem Konstruktor eines Dialogelementes zu finden sind (ein übergeordnetes Fenster, einen Ressourcen-Bezeichner, die Dimensionen *x*, *y*, *h* und *w* des Dialogelementes und optional einen Bibliotheks-Bezeichner), erwartet. Zusätzlich dazu erwartet er als Parameter die Eigenschaft und die maximale Textlänge. Der Konstruktor von *TComboBox* ist wie folgt deklariert:

```
TComboBox(TWindow *parent, int id, int x, int y, int w, int h, DWORD
          style, WORD textLen, TLibId libId = 0);
```

Alle Kombinationsfenster haben die Eigenschaften WS_CHILD, WS_VISIBLE, WS_GROUP, WS_TABSTOP, CBS_SORT (zum Sortieren der Listeneinträge), CBS_AUTOHSCROLL (damit der Benutzer mehr Text eingeben kann als im sichtbaren Editierbereich hineinpaßt) und WS_VSCROLL (vertikale Bildlaufleiste). Der von Ihnen bereitgestellte Eigenschaftsparameter ist einer der Kombinationsfenstereigenschaften von Windows CBS_SIMPLE, CBS_DROPDOWN oder CBS_DROPDOWNLIST. Die Textlänge gibt die maximale Zahl von Zeichen an, die im Editierfeld erlaubt sind.

Die folgenden Zeilen zeigen einen typischen Konstruktoraufruf eines Kombinationsfensters, der eine aufklappbares Kombinationsfenster mit einer unsortierten Liste konstruiert:

```
Combo1 = new TComboBox(this, ID_COMBO1, 190, 30, 150, 100, CBS_SIMPLE, 20);
Combo1->Attr.Style &= ~CBS_SORT;
```

Ändern von Kombinationsfenstern

TComboBox definiert mehrere Elementfunktionen für das Ändern von Listen und Editierfeldern eines Kombinationsfensters. Die unten stehende Tabelle faßt diese Funktionen zusammen.

Da *TComboBox* von *TListBox* abgeleitet ist, können Sie die Elementfunktionen von *TListBox* verwenden, um den Listenbereich eines Kombinationsfensters zu manipulieren.

Tabelle 12.12: Elementfunktionen von *TComboBox* für das Ändern eines Kombinationsfensters

Beschreibung	Elementfunktion
SetText	Ersetze den gesamten Text im Editierbereich
SetEditSel	Wähle Text im Editierbereich aus
Clear	Lösche gesamten Text im Editierbereich
ShowList oder *ShowList*(TRUE)	Zeige den Listenbereich
HideList oder *ShowList*(FALSE)	Verberge den Listenbereich
SetExtendedUI	Setze die erweiterte Kombinationsfenster-UI

Abfragen der Kombinationsfenster

TComboBox fügt mehrere Elementfunktionen zu denen von *TListBox* geerbten hinzu. Dazu zählen Elementfunktionen für die Abfrage des Inhalts eines Editier- und Listenbereiches in einem Kombinationsfenster. Die unten stehende Tabelle faßt diese Elementfunktionen zusammen.

Tabelle 12.13: Elementfunktionen von *TComboBox* für das Abfragen eines Kombinationsfensters

Beschreibung	Elementfunktion
GetTextLen	Gib die Textlänge im Editierbereich zurück
GetText	Rufe den gesamten Text im Editierbereich ab
GetEditSel	Gib die Indizes das ausgewählten Textes im Editierbereich zurück
GetDroppedControlRect	Gib Rechteck der aufgeklappten Liste zurück
GetDroppedState	Bestimme, ob der Listenbereich sichtbar ist
GetExtendedUI	Bestimme, ob das Kombinationsfenster eine erweiterte UI hat

Einstellen und Lesen von Dialogelementwerten

Um komplexe Dialogfenster oder Fenster mit vielen untergeordneten Dialogelemente zu verwalten, müßten Sie typischerweise eine abgeleitete Klasse erzeugen, mit deren Hilfe Sie den Zustand der Dialogelemente speichern und abfragen können. Der Zustand eines Dialogelementes schließt den Text eines Editierfeldes, die Position einer Bildlaufleiste und die Frage, ob ein Schaltfeld markiert ist, mit ein.

Warum Transferpuffer verwenden?

Sie können auch den Aufwand der Definition einer abgeleiteten Klasse für den Zustand ihrer Dialogelemente vermeiden. Dazu verwenden Sie alternativ eine Struktur, die die Dialogelementzustände des Dialogfensters und des Fensters repräsentiert. Diese Struktur heißt *Transferpuffer*, da die Dialogelementzustände von den Dialogelementen zum Puffer und vom Puffer zu den Dialogelementen transferiert werden.

Ihre Anwendung kann zum Beispiel ein modales Dialogfenster hervorbringen und, nachdem es der Benutzer schließt, Information vom Transferpuffer über den Zustand von jedem Dialogelement extrahieren. Wenn dann der Benutzer das Dialogfenster wieder hervorbringt, können Sie die Dialogelementzustände vom Transferpuffer auch wieder transferieren. Zusätzlich können Sie den Anfangszustand für jedes Dialogelement, das auf dem Transferpuffer basiert, neu setzen. Sie können auch explizit Daten in jede Richtung transferieren wie beispielsweise beim Zurücksetzen der Dialogelementzustände auf ihre vorherigen Werte. Ein Fenster oder ein nicht-modales Dialogfenster mit Dialogelemente kann ebenso den Transfermechanismus verwenden, um Zustandsinformationen zu jeder Zeit zu setzen abzufragen.

Assoziieren von Dialogelementobjekten mit Dia- logelementen als Oberflächenelementen ist in Kapitel 8 beschrieben.

Der Transfermechanismus erfordert ObjectWindows-Objekte, um die Dialogelemente zu repräsentieren, für die Sie Daten transferieren wollen.

Für den Transfermechanismus müssen Sie drei Schritte durchführen:

- Definieren Sie den Transferpuffer.
- Definieren Sie das korrespondierende Fenster oder Dialogfenster.
- Transferieren Sie die Daten

Definieren des Transferpuffers

Der Transferpuffer ist eine Struktur mit einem Element für jedes Dialogelement, das an dem Transfer teilnimmt. Ein Fenster oder ein Dialogfenster kann auch Dialogelemente besitzen, die keinen Zustand zu transferieren haben. so nehmen zum Beispiel Aktionsschalter, Gruppenfelder und statische Dialogelemente standardmäßig nicht am Transfer teil.

Der Dialogelementtyp bestimmt den im Transferpuffer benötigten Elementtyp.

Für die Definition eines Transferpuffers, müssen Sie für jedes teilnehmende Dialogelement im Dialogfenster oder Fenster ein Element definieren. Es ist nicht notwendig, für jedes Dialogelement ein Element zu definieren, nur bei solchen Dialogelementen, für die Sie einen Transfer wollen. Der folgende Transferpuffer speichert jeden Dialogelementtyp einmal mit Ausnahme von Aktionsschaltern, Gruppenfeldern und statischen Dialogelementen. Zum Beispiel:

```
struct TSampleTransferStruct {
  char editCtl[sizeOfEditCtl];  // Editierfeld
  WORD checkBox;                // Markierungsfeld
  WORD radioButton;             // Schaltfeld
  TListBoxData *listBox;        // Listenfeld
  TComboBoxData *comboBox;      // Kombinationsfenster
  TScrollBarData *scrollBar;    // Bildlaufleiste
  };
```

Bei jedem Dialogelementtyp muß eine andere Information gespeichert werden. Die unten stehende Tabelle erklärt den Transferpuffer für jedes Dialogelement von ObjectWindows.

Tabelle 12.14: Transferpufferelemente für jeden Dialogelementtyp

Dialogelementtyp	Type	Beschreibung
statisches Feld	**char** array	Ein Zeichen-Array bis zur maximal erlaubten Textlänge zuzüglich der terminierenden NULL. Standardmäßig nehmen statische Dialogelemente nicht am Transfer teil, aber Sie können dies explizit einstellen.
Editierfeld	**char** array	Ein Zeichen-Array bis zur maximal erlaubten Textlänge zuzüglich der terminierenden NULL.
Listenfeld	*TListBoxData**	Ein Zeiger auf eine Instanz der Klasse *TListBoxData*; *TListBoxData* hat mehrere Elemente für die Aufnahme der Listenfeld-Strings, Listeneintragsdaten und für die ausgewählten Indizes.
Kombinationsfenster	*TComboBoxData**	Ein Zeiger auf eine Instanz der Klasse *TComboBoxData*; *TComboBoxData* hat mehrere Elemente für die Aufnahme der Listenbereich-Strings des Kombinationsfensters, der Listeneintragsdaten, der Indexauswahl und des Editierbereichinhalts.
Markierungsfeld Schaltfeld	*WORD*	BF_CHECKED, BF_UNCHECKED, BF_GRAYED; sie zeigen den Zustand des Auswahlfeldes an.

Bildlaufleiste	*TScrollBarData**	Ein Zeiger auf eine Instanz der Klasse *TScrollBarData*; *TScrollBarData* besitzt drei **int**-Elemente: *LowValue* für die Bereichsuntergrenze, *HighValue* für die Bereichsobergrenze und *Position*, für die aktuelle Stellung des Positionszeigers

Transfer von Listenfeldern

Da Listenfelder mehrere Informationsteile transferieren müssen (Strings, Listeneintragsdaten und Auswahlindizes), verwendet der Transferpuffer eine Klasse mit Namen *TListBoxData*. *TListBoxData* besitzt mehrere Datenelemente für die Aufnahme der Listenfeldinformation:

Tabelle 12.15: Datenelemente von *TListBoxData*

Datenelement	Type	Beschreibung
Strings	*TStringArray**	Enthält alle Strings des Listenfeldes
ItemDatas	*TDwordArray**	Enthält die Listeneintragsdaten *DWORD* für jeden Eintrag im Listenfeld
SelIndices	*TIntArray**	Enthält die Indizes aller ausgewählten Strings (in einem Listenfeld mit mehrfacher Auswahl)

TListBoxData besitzt ferner mehrere Elementfunktionen zur Manipulation der Listenfelddaten:

Tabelle 12.16: Elementfunktionen von *TListBoxData*

Elementfunktion	Beschreibung
AddString	Fügt einen String zum Array *Strings* hinzu und wählt ihn optional aus.
AddItemData	Fügt die Daten für einen Listeneintrag zum Array *ItemDatas* hinzu.
AddStringItem	Fügt einen String zum Array *Strings* hinzu, wählt ihn optional aus und fügt die Daten für einen Listeneintrag zum Array *ItemDatas* hinzu.
Select	Wählt einen String zum gegebenen Index aus.
SelectString	Wählt den gegebenen String aus.
ResetSelections	Entfernt alle Auswahlen vom Array *SelIndices*.
GetSelStringLength	Gibt die Länge des ausgewählten Strings am gegebenen Index zurück.
GetSelString	Gibt den ausgewählten String am gegebenen Index zurück.

Transfer von Kombinationsfenstern

Da Kombinationsfenster mehrere Informationsteile transferieren müssen (Strings, Listeneintragsdaten, ausgewählter Eintrag und der Index des ausgewählten Eintrags), verwendet der Transferpuffer eine Klasse mit Namen *TComboBoxData*. *TComboBoxData* besitzt mehrere Datenelemente für die Aufnahme der Information zu Kombinationsfenstern:

Tabelle 12.17: Datenelemente von *TComboBoxData*

Datenelement	Type	Beschreibung
Strings	*TStringArray**	Enthält alle Strings des Listenfeldes
ItemDatas	*TDwordArray**	Enthält die Listeneintragsdaten *DWORD* für jeden Eintrag im Listenfeld
Selection	**Char***	Enthält den ausgewählten String

TComboBoxData besitzt ebenso mehrere Elementfunktionen für die Manipulation der Informationen von Kombinationsfenstern:

Tabelle 12.18: Elementfunktionen von *TComboBoxData*

Elementfunktion	Beschreibung
AddString	Fügt einen String zum Array *Strings* hinzu und wählt ihn optional aus.
AddItemData	Fügt die Daten für einen Listeneintrag zum Array *ItemDatas* hinzu.
AddStringItem	Fügt einen String zum Array *Strings* hinzu, wählt ihn optional aus und fügt die Daten für einen Listeneintrag zum Array *ItemDatas* hinzu.

Definieren des korrespondierenden Fensters oder Dialogfensters

Ein Fenster oder Dialogfenster, das den Transfermechanismus verwendet, muß seine teilnehmenden Dialogelementobjekte in der gleichen Ordnung konstruieren, in der die Elemente des korrespondierenden Transferpuffers definiert sind. Um den Transfer für ein Fensterobjekt oder Dialogfensterobjekt zu ermöglichen, müssen Sie *SetTransferBuffer* aufrufen und einen Zeiger auf den Transferpuffer übergeben.

Verwenden von Transfer bei einem Dialogfenster

Da Dialogfenster ihre Definitionen und die Definitionen ihrer Dialogelemente von Ressourcen erhalten, sollten Sie Dialogelementobjekte über Konstruktoren, die eine Ressource-ID erwarten, konstruieren. Ein Beispiel:

```
struct TTransferBuffer {
  char edit[30];
  TListBoxData *listBox;
  TScrollBarData *scrollBar;
}
  .
  .
  .
TTransferDialog::TTransferDialog(TWindow* parent, int resId)
  : TDialog(parent, resId),
    TWindow(parent)
{
  new TEdit(this, ID_EDIT, 30);
  new TListBox(this, ID_LISTBOX);
  new TScrollBar(this, ID_SCROLLBAR);

  SetTransferBuffer(&TTransferBuffer);
}
```

Dialogelementobjekte, die Sie auf solche Weise konstruieren, haben den Transfer automatisch eingeschaltet (außer für Aktionsschalter, Gruppenfelder und statische Dialogelementobjekte). Um ein Dialogelement explizit vom Transfermechanismus auszuschließen, müssen Sie seine Elementfunktion *DisableTransfer* nach seinem Konstruieren aufrufen.

Verwenden von Transfer bei einem Fenster

Die in einem Fenster konstruierten Dialogelemente haben den Transfer standardmäßig abgeschaltet. Um den Transfer anzuschalten, müssen Sie die Elementfunktion *EnableTransfer* des Dialogelementobjektes aufrufen:

```
ListBox = new TListBox(this, ID_LISTBOX, 20, 20, 340, 100);
ListBox->EnableTransfer();
```

Transferieren der Daten

In den meisten Fällen läuft das Transferieren von Daten zum oder vom Fenster automatisch ab. Sie können aber auch Daten zu jeder Zeit explizit transferieren.

Transferieren von Daten zu einem Fenster

Der Transfer zu einem Fenster passiert automatisch, wenn Sie ein Fensterobjekt konstruieren. Der Konstruktor ruft *SetupWindow* auf, um ein Oberflächenelement zu erzeugen, das das Fensterobjekt repräsentiert. Danach ruft er *TransferData* auf, um alle Daten vom Transferpuffer zu laden. Die Funktion *SetupWindow* des Fensterobjektes ruft auch *SetupWindow* für jedes untergeordnete Fenster auf. Damit hat jedes untergeordnete Fenster die Chance, seine Daten zu transferieren. Da das übergeordnete Fenster seine untergeordneten Fenster in der Reihenfolge einrichtet, in der es sie auch konstruiert, müssen die Daten im Transferpuffer in derselben Reihenfolge erscheinen.

Transferieren von Daten von einem Dialogfenster

Wenn ein modales Dialogfenster eine Kommandobotschaft mit der Dialogelement-ID IDOK empfängt, transferiert es automatisch Daten von den Dialogelementen in den Transferpuffer. Im allgemeinen zeigt diese Botschaft an, daß der Benutzer »OK« gewählt hat, um das Dialogfenster zu schließen. Daher aktualisiert das Dialogfenster automatisch seinen Transferpuffer. Wenn Sie danach das Dialogfenster wieder ausführen, transferiert es vom Transferpuffer zu den Dialogelementen.

Transferieren von Daten von einem Fenster

Sie können zu jeder Zeit Daten explizit in jede Richtung transferieren. Sie könnten zum Beispiel einen Transfer von Daten aus den Dialogelemente in einem Fenster oder einem nicht-modalen Dialogfenster benötigen. Oder aber Sie wollen den Zustand der Dialogelemente zurücksetzen, indem Sie die Daten im Transferpuffer verwenden, sobald der Benutzer auf einen »Zurücksetzen«- oder »Zurückkehren«-Schalter klickt.

Verwenden Sie die Elementfunktion *TransferData* in beiden Fällen. Übergeben Sie dazu TF_SETDATA, um vom Transferpuffer zu den Dialogelemente zu transferieren, oder TF_GETDATA, um von den Dialogelementen zum Transferpuffer zu transferieren.

Sie könnten zum Beispiel *TransferData* in der Elementfunktion *CloseWindow* eines Fensterobjektes aufrufen:

```
void TMyWindow::CloseWindow()
{
  TransferData(TF_GETDATA);
  TWindow::CloseWindow();
}
```

Unterstützen von Transfer für spezielle Dialogelemente

Eventuell wollen Sie die Art ändern, mit der ein einzelnes Dialogelement seine Daten transferiert, oder ein neues Dialogelement, das Sie im Transfermechanismus definieren, miteinbinden. In beiden Fällen müssen Sie nur eine Elementfunktion *Transfer* für Ihre Dialogelementobjekt schreiben. Siehe die unten stehende Tabelle für eine Interpretation der Bedeutung des Transfer-Flag-Parameters.

Tabelle 12.19: Transfer-Flag-Parameter

Transfer-Flag-Parameter	Beschreibung
tdGetData	Kopiere Daten vom Dialogelement an die Stelle, die durch den bereitgestellten Zeiger angegeben ist. Gib die Zahl der transferierten Bytes zurück.
tdSetData	Kopiere die Daten vom Transferpuffer am bereitgestellten Zeiger in das Dialogelement. Gib die Zahl der transferierten Bytes zurück.
tdSizeData	Gib die Zahl Bytes zurück, die transferiert würden.

K A P I T E L 13

Gadgets und Gadgetfensterobjekte

Dieses Kapitel behandelt die Verwendung von Gadgets und Gadgetfenstern. Gadgets besitzen eine ähnliche Funktionalität wie Dialogelemente, wenn sie für Benutzereingaben oder für Ausgaben von Informationen an den Benutzer verwendet werden. Gadgets werden jedoch anders implementiert als Dialogelemente. Im Gegensatz zu den meisten anderen Oberflächenelementen stellen Gadgets keine Fenster dar. Gadgets besitzen keine Fenster-Handles, sie empfangen keine Nachrichten, reagieren nicht auf Ereignisse und sind nicht von der Klasse *TWindow* abgeleitet.

Statt dessen müssen Gadgets immer Teil eines Gadgetfensters sein. Solch ein Gadgetfenster steuert die Darstellung der Gadgets, den Nachrichtenstrom usw. Das Gadget empfängt seine Anordnungen und Richtlinien vom Gadgetfenster.

Im vorliegenden Kapitel werden die verschiedenen Gadgettypen erläutert, die in ObjectWindows 2.5 implementiert sind. Danach erfolgt eine Beschreibung der verschiedene Gadgetfenstertypen, in denen Gadgets eingesetzt werden können.

Gadgets

Im vorliegenden Abschnitt werden einige Gadgets vorgestellt. Zunächst erfolgt eine Erläuterung von *TGadget*, der Basisklasse für Gadgets in ObjectWindows. Danach werden die anderen Gadgetklassen *TSeparatorGadget*, *TBitmapGadget*, *TControlGadget*, *TTextGadget* und *TButtonGadget* besprochen.

Die Klasse TGadget

Alle Gadgets basieren auf der Klasse *TGadget*. Diese Klasse enthält grundlegende Funktionen, die von allen Gadgets benötigt werden, einschließlich der Funktionen zur Steuerung des Gadgetrahmens, der Rah-

menart, zur Einstellung der Größe eines Gadget, zum Aktivieren und Deaktivieren eines Gadget, usw.

Erzeugen und Löschen einer Instanz der Klasse TGadget

Ein *TGadget*-Konstruktor besitzt folgende Syntax:

```
TGadget(int id = 0, TBorderStyle style = None);
```

Dabei ist:

- *id* ein willkürlicher Wert, der die ID-Nummer des Gadget darstellt. Mit Hilfe dieser ID können Sie ein bestimmtes Gadget innerhalb eines Gadgetfensters ansprechen. Weitere Verwendungsmöglichkeiten für die ID eines Gadget finden Sie im nächsten Abschnitt.
- style ein **enum** *TBorderStyle*. Dabei kann *style* fünf verschiedene Werte annehmen:

  None : das Gadget wird rahmenlos dargestellt; das heißt, der Rahmen ist nicht sichtbar.

  Plain : der Rahmen des Gadget ist wie der Rand eines Fensterrahmens als Linie sichtbar.

  Raised : das Gadget sieht aus, als würde es sich vom Gadgetfenster abheben.

  Recessed : das Gadget sieht aus, als würde es im Gadgetfenster versinken.

  Embossed : das Gadget sieht aus, als besäße es einen herausgearbeiteten Grat als Rahmen.

Der *TGadget*-Destruktor wird **virtual** deklariert. Er entfernt das Gadget aus dem Gadgetfenster, wenn das Fenster noch aktiv ist.

Identifizieren eines Gadgets

Sie können ein Gadget mit Hilfe der Funktion *GetId* identifizieren. Diese Funktion gibt die ID des Gadget aus. *GetId* erwartet keine Parameter und liefert einen **int** als Rückgabewert, der der ID des Gadget entspricht. Diese ID ist der Wert, der zuvor als erster Parameter an den *TGadget*-Konstruktor übergeben wurde.

Es gibt mehrere Möglichkeiten für die Nutzung der ID eines Gadget:

- Mit Hilfe der ID können Sie ein bestimmtes Gadget identifizieren. Wenn Sie viele Gadgets innerhalb eines Gadgetfensters nutzen, dann ist dies der einfachste Weg, sich zwischen den einzelnen Gadgets zurechtzufinden.
- Wenn das Gadget zur Erzeugung eines Befehls verwendet wird, dann können Sie die ID so setzen, daß sie der ID des gewünschten Ereignis-

ses gleicht. Zum Beispiel besitzt eine Schaltervorrichtung, die zum Öffnen einer Datei verwendet wird, die ID CM_FILEOPEN.

- Wenn in einer Meldungszeile oder Statuszeile ein Textstring angezeigt werden soll, sobald das Gadget angeklickt wird, dann können Sie einen String-Bezeichner als ID angeben. Nehmen wir beispielsweise an, Sie hätten den String-Bezeichner IDS_MYSTRING definiert, über den ein Gadget beschrieben wird. Sie können nun die ID des Gadget auf IDS_MYSTRING setzen. Wenn Ihr Fenster eine Meldungs- oder Statuszeile hat und Sie Menümitverfolgung gewählt haben, wird der String IDS_MYSTRING in der Meldungs- oder Statuszeile angezeigt, sobald Sie das Gadget IDS_MYSTRING anklicken.

Die letzten beiden Techniken werden häufig kombiniert. Angenommen, Sie haben für den Menübefehl Datei Öffnen einen Kommandobezeichner CM_FILEOPEN definiert, dann können Sie auch dem Gadget den Bezeichner CM_FILEOPEN zuordnen. Wird danach das Gadget angeklickt, sendet das Gadgetfenster ein CM_FILEOPEN-Ereignis. Gibt es nun einen String mit der Ressourcen-ID CM_FILEOPEN, dann wird dieser String in der Meldungs- oder Statuszeile angezeigt, sobald das Gadget angeklickt wird. Eine Illustration dieser Technik finden Sie in Schritt 10 des Kapitels 2.

Verändern und Zugreifen auf das Erscheinungsbild eines Gadget

Mit Hilfe der folgenden Funktionen können Sie die Seitenrandbreite, die Rahmenbreite und die Rahmenart eines Gadget überprüfen und verändern:

```
void SetBorders(TBorders& borders);
TBorders &GetBorders();
void SetMargins(TMargins& margins);
TMargins &GetMargins();
void SetBorderStyle(TBorderStyle style);
TBorderStyle GetBorderStyle();
```

Der Rahmen stellt die äußerste Begrenzung eines Gadget dar. Die Struktur *TBorders* wird als Parametertyp in den Funktionen *SetBorders* und *GetBorders* verwendet und besitzt vier Datenelemente. Diese vorzeichenlosen (**unsigned**) Datenelemente *Left, Right, Top* und *Bottom* enthalten die Breite des jeweiligen Rahmen eines Gadget.

Der Seitenrand ist der Bereich zwischen dem Rahmen und dem inneren Rechteck eines Gadget. Die Struktur *TMargins*, die als Parametertyp in den Funktionen *SetMargins* und *GetMargins* verwendet wird besitzt ebenfalls vier Datenelemente. Diese **int**-Datenelemente *Left, Right, Top* und *Bottom* enthalten die Breite des jeweiligen Seitenrandes eines Gadget.

Der Parametertyp *TBorderStyle* wird in den Funktionen *SetBorderStyle* und *GetBorderStyle* verwendet und entspricht dem Parametertyp, der im

TGadget-Konstruktor benutzt wird. Die Effekte der verschiedenen Rahmenarten werden dadurch erreicht, daß die Seitenränder und die Seiten des Gadgetrahmens für jede Rahmenart unterschiedlich gezeichnet werden.

Begrenzen eines Gadget

Den gesamten Bereich, der vom Gadget belegt wird, nennt man begrenzendes Rechteck. Er ist in der Struktur *TRect* enthalten und setzt sich aus den relativen X- und Y-Koordinaten der oberen linken und der unteren rechten Ecke des Gadget innerhalb des Gadgetfensters zusammen. Mit Hilfe des begrenzenden Rechtecks wird ein Gadget in einem Gadgetfenster positioniert. Weiterhin ist das begrenzende Rechteck wichtig für die Überprüfung, ob der Benutzer das Gadget angeklickt hat.

Das begrenzende Rechteck eines Gadget kann über folgende Funktionen abgefragt und eingerichtet werden:

```
TRect &GetBounds();
virtual void SetBounds(TRect& rect);
```

Beachten Sie, daß die Funktion *SetBounds* **virtual** deklariert wird. Standardmäßig aktualisiert *SetBounds* nur die Daten des begrenzenden Rechtecks. Eine abgeleitete Klasse kann *SetBounds* überschreiben, um Veränderungen gleichzeitig anzuzeigen und den internen Status eines Gadget zu aktualisieren.

Größenanpassung für ein Gadget

Sie können angeben, ob ein Gadgetfenster ein Gadget über die Funktion *SetShrinkWrap* größenmäßig anpassen soll oder nicht. Wurde die Größenanpassung für eine Achse aktiviert, dann wird die Gesamtgröße eines Gadget auf der Grundlage der Rahmenbreite, der Seitenrandbreite und der Größe des inneren Rechtecks automatisch berechnet. Das bedeutet, daß eine manuelle Berechnung der äußeren Größe des Gadget nicht nötig ist.

Sie können die Größenanpassung für die Breite und für die Höhe eines Gadget unabhängig von einander ein- und ausschalten:

```
void SetShrinkWrap(BOOL shrinkWrapWidth, BOOL shrinkWrapHeight);
```

Hierbei schaltet:

- die Funktion *shrinkWrapWidth* die horizontale Anpassung ein- oder aus, je nachdem, ob TRUE oder FALSE übergeben wurde.
- die Funktion *shrinkWrapHeight* die vertikale Anpassung ein- oder aus, je nachdem, ob TRUE oder FALSE übergeben wurde.

Festlegen der Gadgetgröße

Die Größe eines Gadget ist die Größe des begrenzenden Rechtecks dieses Gadget. Die Größe unterscheidet sich vom begrenzenden Rechteck nur dadurch, daß sie nicht von der Position des Gadget abhängig ist. Daher

können Sie die Größe eines Gadget einstellen, ohne daß die Position des Gadget verändert wird.

Mit Hilfe der Funktion *SetSize* legen Sie die gewünschte Größe für ein Gadget fest:

```
void SetSize(TSize& size);
```

Über die Funktion *GetDesiredSize* können Sie sich die aktuelle Größe des Gadget ausgeben lassen:

```
virtual void GetDesiredSize(TSize& size);
```

Auch wenn Sie die für ein bestimmtes Gadget gewünschte Größe mit Hilfe der Funktion *SetSize* gesetzt haben, sollten Sie nachträglich die Funktion *GetDesiredSize* aufrufen, um die wirkliche Größe des Gadget zu überprüfen. Während eines Layoutprozesses kann es vorkommen, daß Gadgetfenster die gewünschte Größe eines Gadget verändern.

Anpassen der Farben des Gadget an die Systemfarben

Wollen Sie Ihre Oberfläche an das Benutzersystem Ihrer Anwendung anpassen, dann sollten Sie die Funktion *SysColorChange* implementieren. Das Gadgetfenster ruft die Funktion *SysColorChange* für jedes im Fenster enthaltene Gadget auf, wenn es die Nachricht WM_SYSCOLORCHANGE erhält. Diese Funktion besitzt folgende Syntax:

```
virtual void SysColorChange();
```

Standardmäßig löst die Funktion *SysColorChange* keine Aktion aus. Sollen sich Ihre Gadgets jedoch nach den Änderungen in den Systemfarben richten, dann sollten Sie diese Funktion implementieren. Dabei müssen Sie unbedingt darauf achten, daß alle Ressourcen gelöscht und freigegeben sind, die von den Einstellungen der Systemfarben abhängig sind.

Public-Datenelemente der Klasse TGadet

In *TGadget* gibt es zwei **public**-Datenelemente; beide sind vom Datentyp BOOL:

```
BOOL Clip;
BOOL WideAsPossible;
```

Der Wert von *Clip* gibt an, ob ein clipping-Rechteck vor der Bildschirmausgabe des Gadget verwendet werden soll.

Der Wert von *WideAsPossible* gibt an, ob das Gadget so erweitert werden soll, daß es den im Fenster verfügbaren Raum ausfüllt. Das erweist sich zum Beispiel für ein Textgadget in einer Meldungszeile als günstig.

Aktivieren und Deaktivieren von Gadgets

Über folgende Funktionen können Sie ein Gadget aktivieren oder deaktivieren:

```
virtual void SetEnabled(BOOL);
BOOL GetEnabled();
```

Wird der Status eines Gadget über die Funktion *SetEnabled* verändert, dann wird danach das begrenzende Rechteck des Gadget zwar für ungültig erklärt, aber nicht entfernt. Eine abgeleitete Klasse kann *SetEnabled* überschreiben um dieses Verhalten zu ändern.

Wird durch ein Gadget ein Kommando erzeugt, dann sollten Sie die Funktion *CommandEnable* implementieren.

```
virtual void CommandEnable();
```

Standardmäßig löst die Funktion *CommandEnable* keine Aktionen aus. Eine abgeleitete Klasse kann das Verhalten dieser Funktion so überschreiben, daß die Aktivierung von Kommandos aktiviert wird. Das Gadget muß die WM_COMMAND_ENABLE Botschaft an das übergeordnete Fenster des Gadgetfensters senden. In dieser Botschaft sollte ein Kommando-Freigabe-Objekt enthalten sein, wobei das Objekt das Gadget repräsentiert.

Im folgenden Programmbeispiel wird die Implementierung der Funktion *CommandEnable* dargestellt:

```
void
TMyGadget::CommandEnable()
{
   Window->Parent->HandleMessage(
      WM_COMMAND_ENABLE,
      0,
      (LPARAM) &TMyGadgetEnabler(*Window->Parent, this));
}
```

Ableiten von TGadget

TGadget enthält mehrere **protected**-Zugriffsfunktionen, die zum Ableiten einer Gadgetklasse von*TGadget* verwendet werden können.

Initialisieren und Entfernen

TGadget bietet viele **virtuelle** Funktionen für das Initialisieren und Entfernen eines Gadget:

```
virtual void Inserted();
virtual void Removed();
```

Die Funktion *Inserted* wird aufgerufen, nachdem ein Gadget in ein Gadgetfenster eingefügt wurde. Die Funktion *Removed* wird aufgerufen, bevor ein Gadget aus seinem Gadgetfenster entfernt wird. Standardmäßig lösen beide Funktionen keine Aktionen aus.

Zeichnen eines Gadget

In der *TGadget*-Klasse sind zwei verschiedene Zeichenfunktionen enthalten: *PaintBorder* und *Paint*.

Die Funktion *PaintBorder* zeichnet den Rahmen eines Gadget. Dieser virtuellen Funktion wird als einziger Parameter ein *TDC* **&** übergeben. Ihr Rückgabewert ist **void**. *PaintBorder* implementiert die Standard-Rahmenarten. Wollen Sie einen neuen Rahmen erzeugen, dann müssen Sie diese Funktion überschreiben und so die Funktionalität für den neuen Rahmen zur Verfügung stellen. Wollen Sie die standardmäßigen Rahmenarten ebenfalls beibehalten, dann müssen Sie auch die *TGadget*-Version dieser Funktion aufrufen. *PaintBorder* wird von der Funktion *Paint* aufgerufen.

Die Funktion *Paint* ähnelt der *TWindow*-Funktion *Paint*. Dieser Funktion wird als einziger Parameter ein *TDC* **&** übergeben. Der Rückgabewert ist **void**. *Paint* ist als **virtual** deklariert. Die *TGadget*-Funktion *PaintGadgets* von *TGadget* ruft beim Zeichnen eines Gadgetfensters für jedes Gadget die *Paint*-Funktion auf. Die standardmäßige Funktion *Paint* ruft lediglich die Funktion *PaintBorder* auf. Soll das innere Rechteck des begrenzenden Rechtecks für das Gadget gezeichnet werden, dann sollten Sie diese Funktion so überschreiben, daß die erforderliche Funktionalität gegeben ist.

Wollen Sie das Gadget über die Funktion *Paint* zeichnen, dann müssen Sie häufig den Bereich innerhalb des Rahmens und der Seitenränder des Gadgets ermitteln. Dieser Bereich wird inneres Rechteck genannt. Sie können sich das innere Rechteck auch über die Funktion *GetInnerRect* ausgeben lassen:

```
void GetInnerRect(TRect& rect);
```

GetInnerRect legt die Koordinaten des inneren Rechtecks in die übergebene Referenz *rect* ab.

Ungültig Erklären und Aktualisieren eines Gadgets

So wie ein Fenster kann auch ein Gadget für ungültig erklärt werden. *TGadget* bietet zwei Funktionen für das Deaktivieren eines Gadget:

```
void Invalidate(BOOL erase = TRUE);
void InvalidateRect(const TRect& rect, BOOL erase = TRUE);
```

Diese Funktionen ähneln den *TWindow*-Funktionen *InvalidateRect* und *Invalidate*. Die Funktion *InvalidateRect* besitzt fast das gleiche Aussehen wie ihre Windows-API-Version, mit der Ausnahme, daß der Parameter HWND fehlt. Die Funktion *Invalidate* erklärt das gesamte begrenzte Rechteck des Gadget für ungültig. *Invalidate* erwartet genau einen einzigen booleschen Parameter. Über diesen Parameter wird angegeben, ob der ungültige Bereich gelöscht werden soll, wenn er aktualisiert wird. Standardmäßig ist dieser Parameter TRUE. Um den gesamten Bereich eines Gadget zu entfernen, müssen Sie daher lediglich die Funktion *Invalidate* mit dem Wert TRUE oder auch ohne jede Parameterangabe aufrufen.

Damit verwandt ist die Funktion *Update*. Über diese wird versucht, eine sofortige Aktualisierung des Gadget zu erzwingen. *Update* ähnelt der Windows-API-Funktion *UpdateWindow*.

```
void Update();
```

Mausereignisse in einem Gadget

Mausereignisse können innerhalb und außerhalb eines Gadget verfolgt werden. Dies geschieht durch verschiedene "Pseudo-Ereignis-Behandlungsroutinen" der *TGadget*-Klasse. Diese Funktionen sind im Aussehen den Standardfunktionen für die Ereignisbehandlung von ObjectWindows sehr ähnlich. Der Unterschied besteht darin, daß die Namen der Gadgetfunktionen nicht mit dem Präfix *Ev* versehen sind.

Gadgets haben keine Antworttabellen wie andere ObjectWindows-Klassen, weil ein Gadget kein wirkliches Fenster darstellt. Die gesamte Kommunikation eines Gadget mit der Außenwelt erfolgt über das Gadgetfenster. Tritt innerhalb eines Gadgetfensters ein Mausereignis ein, dann versucht das Fenster festzustellen, welches Gadget von dem Ereignis betroffen ist. Über den Aufruf der Funktion *PtIn* können Sie herausfinden , ob ein Ereignis innerhalb eines bestimmten Gadget stattgefunden hat:

```
virtual BOOL PtIn(TPoint& point);
```

Standardmäßig gibt diese Funktion TRUE zurück, wenn sich *point* innerhalb des begrenzenden Rechtecks des Gadget befindet. Sie können diese Funktion auch überschreiben, um damit andere Gadget-Formen zu unterstützen.

Sobald die Maus in den zulässigen Bereich eines Gadget eindringt, ruft das Gadgetfenster die Funktion *MouseEnter* auf. Diese Funktion besitzt folgende Syntax:

```
virtual void MouseEnter(UINT modKeys, TPoint& point);
```

modKeys enthält virtuelle Tasteninformationen, die den Informationen entsprechen, die an die Standardfunktion *EvMouseMove* von ObjectWindows übergeben wurden. Dieser Wert gibt an, ob verschiedene virtuelle Tasten gedrückt wurden. Der Parameter kann eine beliebige Kombination folgender Werte enthalten: MK_CONTROL, MK_LBUTTON, MK_MBUTTON, MK_RBUTTON oder MK_SHIFT. Eine vollständige Erläuterung dieser Flags finden Sie im *ObjectWindows Referenzhandbuch*. *point* teilt dem Gadget mit, ob sich die Maus innerhalb des Gadget befindet.

Sobald das Gadgetfenster die Funktion *MouseEnter* aufgerufen hat, um das Gadget darüber zu informieren, daß sich der Mauszeiger im Bereich des Gadget befindet, erfaßt das Gadget alle Mausbewegungen. Das geschieht über den Aufruf der Funktion *GadgetSetCapture* des Gadgetfenster. Das Gadgetfenster wiederum garantiert, daß die Gadgetfunktion *MouseLeave* aufgerufen wird.

Das Gadgetfenster aktiviert die Funktion *MouseLeave,* wenn der Mauszeiger die Grenzen des Gadget verläßt. Diese Funktion besitzt folgendes Aussehen:

```
virtual void MouseLeave(UINT modKeys, TPoint& point);
```

Weiterhin gibt es zwei Funktionen zum Erkennen der Aktivierung der linken Maustaste: *LButtonDown* und *LButtonUp*. Wenn das boolesche Flag *TrackMouse* gesetzt ist, dann erfaßt *LButtonDown* die Maus. Standardmäßig setzt *LButttonUp* die Maus frei, wenn das boolesche Flag *TrackMouse* angegeben wurde. *TrackMouse* ist standardmäßig nicht gesetzt.

```
virtual void LButtonDown(UINT modKeys, TPoint& point);
virtual void LButtonUp(UINT modKeys, TPoint& point);
```

Wird die Maus innerhalb des begrenzenden Rechtecks eines Gadget verschoben, während die Mausbewegungen durch das Gadgetfenster aufgezeichnet werden, dann wird die Gadgetfunktion *MouseMove* durch das Fenster aufgerufen. Diese Funktion besitzt folgende Syntax:

```
virtual void MouseMove(UINT modKeys, TPoint& point);
```

Wie die Funktion *MouseEnter* enthält *modKeys* virtuelle Tasteninformationen. *point* teilt dem Gadget mit, an welcher Position die Bewegung beendet wurde.

Gadgetklassen in ObjectWindows

ObjectWindows enthält viele Klassen, die von *TGadget* abgeleitet wurden Diese Gadgets bieten vielseitige und einfach zu nutzende Dekorationen sowie neue Möglichkeiten zur Kommunikation mit dem Benutzer Ihrer Anwendung. Folgende Gadgetklassen sind in ObjectWindows enthalten:

- *TSeparatorGadget*
- TTextGadget
- TButtonGadget
- TControlGadget
- TBitmapGadget

In den folgenden Abschnitten werden die oben aufgeführten Klassen erläutert.

Die Klasse TSeparatorGadget

TSeparatorGadget stellt ein sehr einfaches Gadget dar. Seine einzige Aufgabe besteht darin, einen Bereich des Gadgetfensters in Anspruch zu nehmen. Sie können diese Klasse benutzen, wenn in einem Fenster bereits andere Gadgets angeordnet sind. Damit wird erreicht, daß zwischen den Gadgets ein Zwischenraum freibleibt. Ansonsten würden die Gadgets in dem Fenster direkt nebeneinander Wand an Wand angeordnet werden.

Der Konstruktor *TSeparatorGadget* besitzt folgendes Aussehen:

```
TSeparatorGadget(int size = 6);
```

Der Separator deaktiviert sich selbst und schaltet die Größenanpassung aus. Der Parameter *size* wird sowohl für die Breite als auch für die Höhe des Gadget verwendet. Dadurch können Sie den Separatorgadget sowohl für vertikale als auch für horizontale Zwischenräume verwenden.

Die Klasse TTextGadget

Mit Hilfe der Klasse *TTextGadget* werden in einem Gadgetfenster Textinformationen angezeigt. Sie können angeben, wieviele Zeichen in dem Gadget angezeigt werden dürfen. Weiterhin können Sie angeben, wie der Text in dem Textgadget ausgerichtet werden soll.

Erzeugen und Löschen einer Instanz der Klasse TTextGadget

Es folgt die Syntax für den Konstruktor von *TTextGadget*:

```
TTextGadget(int id = 0,
            TBorderStyle style = Recessed,
            TAlign alignment = Left,
            UINT numChars = 10,
            const char* text = 0);
```

wobei:

- *id* die Gadget-ID darstellt.
- style die Rahmenart für das Gadget angibt.
- align festlegt, wie der im Gadget enthaltene Text ausgerichtet werden soll. Für **enum** *TAlign* sind drei Werte zulässig: *Left, Center* und *Right*.
- numChars die Anzahl der Zeichen spezifiziert, die in dem Gadget angezeigt werden. Über diesen Parameter wird die Breite des Gadget festgelegt. Das Gadget berechnet die erforderliche Gadgetbreite durch Multiplizieren der Zeichenanzahl mit der maximalen Zeichenbreite der aktuellen Schrift. Die Höhe des Gadget beruht auf der maximalen Zeichenhöhe der aktuellen Schrift plus dem Zwischenraum für den Seitenrand und den Rahmen.
- text ist eine Standardmeldung, die im Gadget ausgegeben werden soll.

~TTextGadget gibt den Speicherplatz für den Textstring des Gadget automatisch frei.

Zugreifen auf den Gadgettext

Sie können sich den im Gadget enthaltenen Text über die Funktion *GetText* ausgeben lassen und über die Funktion *SetText* neu festlegen.

GetText erwartet keine Parameter und gibt **const char *** zurück. Sie sollten den Gadgettext möglichst nicht über den zurückgegebenen Zeiger modifizieren.

SetText erwartet einen **const char * -**Zeiger übergeben werden und gibt **void** (also nichts) zurück. Das Gadget erstellt eine Kopie des Textes und speichert diese intern ab.

Die Klasse TBitmapGadget

TBitmapGadget ist ein einfaches Gadget, über das ein Array mit Bitmap-Bildern angezeigt werden kann (immer nur eins gleichzeitig). Sie sollten die Bitmaps als Array speichern. Dafür müssen die einzelnen Bitmaps nebeneinander in einer einzelnen Bitmap-Ressource gezeichnet werden. Hierbei müssen alle Bitmaps die gleiche Breite haben.

Erzeugen und Löschen einer Instanz der Klasse TBitmapGadget

Der Konstruktor für *TBitmapGadget* besitzt folgende Syntax:

```
TBitmapGadget(TResId bmpResId,
              int id,
              TBorderStyle style,
              int numImages,
              int startImage);
```

Dabei ist:

- *bmResId* der Ressourcenbezeichner für die Bitmap-Ressource.
- *id* die Gadget-ID.
- *style* die Rahmenart des Gadget.
- *numImages* die Gesamtanzahl der in der Bitmap enthaltenen Bilder. Das Gadget ermittelt die Breite jeder einzelnen Bitmap in der Ressource, indem es die Breite der Bitmap-Ressource durch *numImages* dividiert.

Angenommen, Sie haben an dem Konstruktor von *TBitmapGadget* eine Bitmap-Ressource übergeben, die 400 Pixel breit und 200 Pixel hoch ist. Weiterhin haben Sie den Wert 4 für *numImages* angegeben. Dann teilt der Konstruktor die Bitmap-Ressource in vier einzelne Bitmaps, von denen jedes 100 Pixel breit und 200 Pixel hoch ist.

- *startImage* bestimmt, welches Bitmap des Array im Gadget als erstes angezeigt werden soll.

Der Destruktor *~TBitmapGadget* gibt den für die Bitmap-Bilder verwendeten Speicherplatz wieder frei.

Auswählen eines neuen Bildes

Über die Funktion *SelectImage* können Sie das im Gadget angezeigte Bild wechseln:

```
int SelectImage(int imageNum, BOOL immediate);
```

Der Parameter *imageNum* enthält die Array-Indexnummer des Bildes, das im Gadget angezeigt werden soll. Wenn Sie für *immediate* TRUE angeben, dann aktualisiert das Gadget die Anzeige sofort. Ansonsten wird der Bereich für ungültig erklärt und aktualisiert, sobald die nächste Meldung WM_PAINT eintrifft.

Einstellen der Systemfarben

Über die Klasse *TBitmapGadget* wird die Funktion *SysColorChange* implementiert, so daß sich die Bitmaps nach den Systemfarben richten. Die Funktion löscht das Bitmap-Array, ruft die Funktion *MapUIColors* für die Bitmap-Ressource auf und erstellt danach das Array neu. Siehe nähere Informationen über die Funktion *MapUIColors*.

Die Klasse TButtonGadget

Schaltergadgets sind die einzigen in ObjectWindows enthaltenen Gadgettypen, die mit dem Benutzer direkt kommunizieren. Die Dialogelementgadgets, die im nächsten Abschnitt besprochen werden, bieten ebenfalls ein Gadget, das Benutzereingaben entgegennimmt. Dies geschieht jedoch über eine Dialogelement-Klasse. In diesem Fall agiert das Gadget nur als Vermittler zwischen dem Dialogelement und dem Gadgetfenster.

Ein normales Schaltergadget kann drei Zustände annehmen: *up* (oben), *down* (unten) und *indeterminate* (unbestimmt). Außerdem kann der Schalter in allen drei Zuständen hervorgehoben dargestellt werden, wenn er gedrückt wird.

Es gibt zwei grundlegende Schaltergadgets: die Befehlsgadgets und die Einstellungsgadgets. Einstellungsgadgets können exklusiv (wie ein Schaltfeld) oder nicht exklusiv (wie ein Markierungsfeld) sein. Befehlsgadgets können nur den Zustand *up* besitzen, während Einstellungsgadgets alle drei Zustände annehmen können.

Wenn die linke Maustaste gedrückt wird, während sich der Mauszeiger innerhalb des begrenzenden Rechtecks eines Gadget befindet, wird das Schaltergadget aktiviert. Das Gadget wird beim Drücken markiert.

Sobald das Gadget angeklickt wurde erfaßt das Gadget alle Mausbewegungen. Bewegt sich die Maus aus dem begrenzenden Rechteck des Gadget heraus, ohne daß die linke Maustaste wieder freigegeben wird, dann wird das Gadget zwar nicht mehr markiert dargestellt, aber die Mausbewegungen werden weiterhin vom Gadget registriert. Sobald die Maus wieder in das begrenzende Rechteck des Gadget eintritt, ohne daß die linke Maustaste freigegeben wurde, wird das Gadget wieder markiert.

Wird die linke Maustaste freigegeben, dann werden die Mausbewegungen nicht länger erfaßt. Befindet sich der Mauszeiger während der Freigabe der Maustaste innerhalb des zulässigen Bereiches, dann wird die Gadget-ID vom Gadgetfenster als Befehlsmeldung abgesendet.

Erzeugen und Löschen einer Instanz der Klasse TButtonGadget

Der Konstruktor von*TButtonGadget* besitzt folgende Syntax:

```
TButtonGadget(TResId bmpResId,
              int id,
              TType type = Command,
              BOOL enabled = FALSE,
```

```
TState state = Up,
 BOOL repeat = FALSE);
```

wobei:

- das Argument *bmpResId* den Ressourcenbezeichner für die im Schalter anzuzeigende Bitmap enthält. Die Größe der Bitmap bestimmt die Größe des Gadget, weil die Größenanpassung (Shrink Wrapping) aktiv ist.
- id ist die Gadget-ID. Sie entspricht dem Befehl, der versendet wird, sobald das Gadget gedrückt wird.
- type gibt den Gadgettyp an. Für *TType* **enum** sind die folgenden drei Werte zulässig:

Command spezifiziert das Gadget als Befehl,
Exclusive gibt an, daß das Gadget ein exklusiver Einstellungsschalter ist. Exklusive Schaltergadgets, die aneinander angrenzen, arbeiten zusammen. Sie können exklusive Gruppen bilden, indem Sie andere Gadgets, wie Separatorgadgets oder Textgadgets, an jeder Seite der Gruppe einfügen.
NonExclusive gibt an, daß es sich bei dem Gadget um einen nicht-exklusiven Einstellungsschalter handelt.

- *enabled* bestimmt, ob das Schaltergadget sofort nach dessen Erstellung verfügbar ist oder nicht. Ist bei der Erstellung des Gadget das zugehörige Kommando verfügbar, dann wird der Schalter automatisch ebenfalls zur Verfügung gestellt.
- state enthält den Standardzustand des Schaltergadget. **enum** *TState* kann einen der drei folgenden Werte annehmen: *Up*, *Down* oder *Indeterminate*.
- repeat gibt an, ob die mit dem Schalter verbundene Aktion wiederholt wird, wenn der Schalter gedrückt bleibt. Ist *repeat* TRUE, dann wird die Aktion wiederholt.

Die Funktion *~TButtonGadget* löscht die Bitmap-Ressourcen und gibt den Speicherplatz für den String frei, wenn die Ressource Informationen in einem String enthalten waren.

Zugriff auf die Information eines Schaltergadget

Es gibt eine Vielzahl von Funktionen, mit deren Hilfe Sie auf ein Schaltergadget zugreifen können. Über diese Funktionen können Sie den Zustand des Gadget auf einen beliebigen gültigen *TState*-Wert setzen und sich den Zustand und den Typ des Schaltergadget ausgeben lassen.

Über die Funktion *SetButtonState* können Sie den Zustand eines Schaltergadget festlegen:

```
void SetButtonState(TState);
```

Über die Funktion *GetButtonState* können Sie den aktuellen Zustand des Schaltergadget ermitteln:

- TState GetButtonState();

Den Typ eines Schaltergadget bekommen Sie über die Funktion *GetButtonType* heraus:

```
TType GetButtonType();
```

Typ des Schaltergadget festlegen

Mit Hilfe der folgenden Funktionen können Sie das Erscheinungsbild eines Schaltergadget verändern:

- Über die Funktion *SetNotchCorners* können Sie die Einkerbung der Ecken ein- und ausschalten:

```
void SetNotchCorners(BOOL notchCorners=TRUE);
```

- Über die Funktion *SetAntialiasEdges* können Sie die Deaktivierung des Alias für die Schalterabschrägung ein- und ausschalten:

```
void SetAntialiasEdges(BOOL anti=TRUE);
```

Über die Funktion *SetShadowStyle* können Sie die Schattenart des Schalters wechseln . Dafür werden über den **enum** *TShadowStyle* die folgenden zwei Optionen angegeben: *SingleShadow* und *DoubleShadow*.

```
void SetShadowStyle(TShadowStyle style=DoubleShadow);
```

Aktivieren eines Kommandos

TButtonGadget überschreibt die *TGadget*-Funktion *CommandEnable*. Diese wird zum Initiieren einer WM_COMMAND_ENABLE Meldung für das Gadget implementiert.

Es folgt die Syntax der Funktion *TButtonGadget::CommandEnable*:

```
void CommandEnable();
```

Setzen der Systemfarben

TButtonGadget implementiert die Funktion *SysColorChange*, so daß sich die Bitmaps des Gadget nach den Systemfarben richten. Die Funktion baut das Gadget mit Hilfe der Systemfarben erneut auf. Haben sich die Systemfarben geändert, dann spiegeln sich diese Änderungen in dem neuen Schaltergadget wieder. Dies erfolgt *nicht* automatisch; das heißt, die Funktion wird durch ein WM_SYSCOLORCHANGE-Ereignis nicht unbedingt automatisch aufgerufen.

Die Klasse TControlGadget

TControlGadget ist eine einfache Klasse, die als Schnittstelle zwischen einem regulären Windows-Dialogelement (wie einem Schalter, Bearbeitungsfeld, Listenfeld usw.) und einem Gadgetfenster dient. Dadurch können Sie standardmäßige Windows-Dialogelemente, wie Steuerbalken, Statuszeilen usw., in einem Gadgetfenster nutzen.

Erzeugen und Löschen einer Instanz der Klasse TControlGadget

Der Konstruktor der Klasse *TControlGadget* besitzt folgende Syntax:

```
TControlGadget(TWindow& control, TBorderStyle style = None);
```

wobei:

- das Argument *control* eine Referenz auf ein Fensterobjekt von ObjectWindows darstellt. Dieses Objekt muß ein gültiges Dialogelement-Objekt sein.
- style ist die Rahmenart des Gadget.

Die Funktion *~TControlGadget* entfernt das Schnittstellenelement für Dialogelemente und gibt danach den vom Dialogelementobjekt genutzten Speicherplatz frei.

Gadgetfenster

Gadgetfenster basieren auf der Klasse *TGadgetWindow*, die von der Klasse *TWindow* abgeleitet ist. Gadgetfenster sind dazu da, mehrere Gadgets festzuhalten, sie anzuordnen und in einem anderen Fenster anzuzeigen.

Gadgetfenster besitzen ein großes Spektrum an Funktionalität durch die in ihnen enthaltenen Gadgets. Weil Gadgets keine wirklichen Fenster darstellen, können sie keine Ereignismeldungen absenden oder empfangen. Sie können auch nicht direkt mit Fenstern kommunizieren oder selbst Windows-Funktionen aufrufen. Alle vom Gadget zu erfüllenden Aufgaben müssen über das Gadgetfenster erledigt werden.

Ein Gadget besitzt wenig oder gar keine Steuerung darüber, an welcher Stelle des Gadgetfensters es positioniert wird. Das Gadgetfenster ist für die Positionierung und Anordnung aller in ihm enthaltenen Gadgets verantwortlich. Gadgets werden generell in einer Reihe angeordnet, entweder vertikal oder horizontal.

Gadgetfenster treten im allgemeinen nicht einzeln auf, sondern sind gewöhnlich in anderen Fenstern enthalten. Meistens bildet ein dekoriertes Rahmenfenster, wie ein *TDecoratedFrame* oder ein *TDecoratedMDIFrame,* das übergeordnete Fenster zu einem Gadgetfenster. Die Klasse *TToolBox* verwendet allerdings normalerweise ein *TFloatingFrame* als übergeordnetes Fenster.

Erzeugen und Löschen einer Instanz der Klasse TGadgetWindow

Der Konstruktor für die Klasse TGadgetWindow besitzt folgende Syntax:

```
TGadgetWindow(TWindow* parent = 0,
              TileDirection direction = Horizontal,
              TFont* font = new TGadgetWindowFont,
              TModule* module = 0);
```

Dabei ist:

- *parent* ein Zeiger auf das übergeordnete Fensterobjekt.
- direction ein **enum** *TTileDirection*. *direction* kann zwei Werte annehmen: *Horizontal* oder *Vertical*.
- font ein Zeiger auf ein *TFont*-Objekt. Dieses Objekt enthält die Schriftart für das Gadgetfenster. Standardmäßig wird die Schrift auf *TGadgetWindowFont* gesetzt. Dies ist eine Sans-Serif-Schrift variabler Größe, gewöhnlich Helvetica.
- module wird als *TModule*-Parameter an den *TWindow*-Basiskonstruktor übergeben. Standardmäßig ist dieser Parameter gleich 0.

Die Funktion *~TGadgetWindow* entfernt alle im Gadgetfenster enthaltenen Gadgets und löscht danach das Schriftobjekt.

Erstellen eines Gadgetfensters

Die Klasse *TGadgetWindow* überschreibt die Standardelementfunktion *Create* von *TWindow.* Über die *TGadgetWindow*-Version dieser Funktion wird die anfängliche Größe eines Gadgetfensters auf der Grundlage verschiedener Kriterien bestimmt:

- Ob für eines der Gadgets im Fenster Größenanpassung erforderlich ist.
- Die Größe der im Fenster enthaltenen Gadgets.
- Die Richtung der Anordnung im Gadgetfenster.
- Ob das Gadgetfenster einen Rahmen besitzt und wie breit dieser Rahmen ist.

Über die Funktion *Create* wird die richtige Größe des Fensters auf der Grundlage der oben genannten Faktoren bestimmt. Weiterhin werden die Fenstergrößen-Attribute gesetzt und danach *TWindow::Create* aufgerufen, um das Oberflächenelement des Fensters zu erstellen.

Einfügen eines Gadgets in ein Gadgetfenster

Soll ein Gadgetfenster sinnvoll sein, dann muß es einige Gadgets enthalten. Über die Funktion *Insert* werden Gadgets in ein Gadgetfenster eingefügt:

```
virtual void Insert(TGadget& gadget,
                    TPlacement placement = After,
                    TGadget* sibling = 0);
```

wobei:

- *gadget* eine Referenz auf dasjenige Gadget enthalten muß, das in das Gadgetfenster eingefügt werden soll.
- placement angibt, wo das Gadget eingefügt werden soll. Das **enum** *TPlacement* kann die folgenden zwei Werte annehmen: *Before* und *After*. Wird im Parameter *sibling* ein gleichgeordnetes Gadget angegeben, dann wird das Gadget *davor* oder *danach* eingefügt, je nachdem, welcher Wert in *placement* enthalten ist. Ist *sibling* gleich 0, dann wird das

Gadget als erstes Gadget im Fenster eingefügt, wenn *placement* den Wert *Before* enthält, und als letztes Gadget, wenn *placement* gleich *After* ist.

- sibling einen Zeiger auf ein gleichgeordnetes Gadget darstellt.

Wurde das Gadgetfenster bereits erstellt, dann müssen Sie nach der Funktion *Insert* die Funktion *LayoutSession* aufrufen. Eingefügte Gadgets werden erst dann im Fenster sichtbar, wenn das Fenster neu geordnet wurde.

Entfernen eines Gadget aus einem Gadgetfenster

Ein Gadget kann mit Hilfe der Funktion *Remove* aus einem Gadgetfenster entfernt werden:

```
virtual TGadget* Remove(TGadget& gadget);
```

wobei *gadget* eine Referenz auf das Gadget enthält, das aus dem Fenster gelöscht werden soll.

Diese Funktion entfernt das *Gadget* aus dem Gadgetfenster. Zurückgegeben wird ein Zeiger auf das zu entfernende Gadget (*TGadget* *). Das Gadgetobjekt selbst wird nicht gelöscht. *Remove* gibt 0 zurück, wenn sich das angesprochene Gadget nicht im Fenster befindet.

Wie bei *Insert* müssen Sie auch nach der Funktion *Remove* die Funktion *LayoutSession* aufrufen, wenn das Gadgetfenster bereits generiert wurde. Ein gelöschtes Gadget verschwindet erst dann vom Bildschirm, wenn das Fenster neu geordnet wird.

Setzen von Seitenränder und Anordnungsrichtung in einem Fenster

Sowohl vor als auch nach der Generierung des Fensters können Sie die Seitenränder und die Anordnungsrichtung ändern. Dafür verwendet man die Funktionen *SetMargins* und *SetDirection*:

```
void SetMargins(TMargins& margins);
virtual void SetDirection(TTileDirection direction);
```

Beide Funktionen legen die Werte der entsprechenden Datenelemente fest und rufen danach die Funktion *LayoutSession* auf, die im nächsten Abschnitt beschrieben wird).

Mit Hilfe der Funktion *GetDirection* können Sie herausfinden, in welcher Richtung die Gadgets angeordnet sind:

```
TTileDirection GetDirection() const;
```

Anordnen der Gadgets

Soll ein Gadgetfenster neu angeordnet werden, dann müssen Sie die Funktion *LayoutSession* aufrufen.

```
virtual void LayoutSession();
```

Standardmäßig überprüft die Funktion *LayoutSession*, ob bereits Oberflächenelemente im Fenster erstellt wurden. Ist das nicht der Fall, dann wird die Funktion ohne weitere Aktionen beendet. Das Fenster wird auto-

matisch geordnet, sobald ein Fensterelement erstellt wird. Wenn bereits ein oder mehrere Fensterelemente erstellt wurden, dann ordnet *LayoutSession* die Gadgets nebeneinander an und erklärt den modifizierten Bereich des Gadgetfensters für ungültig.

Gewöhnlich wird ein Layoutprozeß durch das Ändern der Seitenränder, das Einfügen bzw. Entfernen von Gadgets oder das Verkleinern bzw. Vergrößern eines Gadget oder Gadgetfensters ausgelöst.

Das eigentliche Nebeneinander-Anordnung der Gadgets übernimmt die Funktion *TileGadgets*:

```
virtual TRect TileGadgets();
```

TileGadgets bestimmt den Platz, der für jedes Gadget benötigt wird, und ordnet die Gadgets der Reihe nach an. Die Funktion liefert ein *TRect* zurück. Dieser Wert enthält den Bereich des Gadgetfensters, der beim Neuordnen der Gadgets verändert wurde.

TileGadgets ruft die Funktion *PositionGadget* auf. Dadurch besteht die Möglichkeit, den Zwischenraum zwischen den Gadgets über abgeleitete Klassen anzupassen, um die Implementierung eines benutzerdefinierten Layout-Schemas zu vereinfachen.

```
virtual void PositionGadget(TGadget* previous, TGadget* next, TPoint&
point);
```

Diese Funktion ermittelt den erforderlichen Abstand zwischen den Gadgets, auf die über *previous* und *next* verwiesen wird, und legt ihn in *point* ab. Erfolgt die Anordnung horizontal, dann ist der benötigte Maßstab in *point.x* enthalten. Erfolgt sie vertikal, dann ist der Maßstab in *point.y* enthalten.

Benachrichtigen des Fensters über die Größenänderung eines Gadget

Wird ein Gadget verkleinert oder vergrößert, dann sollte die Funktion *GadgetChangedSize* für das zugehörige Gadgetfenster aufgerufen werden. Diese Funktion besitzt folgende Syntax:

```
void GadgetChangedSize(TGadget& Gadget);
```

Das Argument *Gadget* ist eine Referenz auf das Gadget, dessen Größe verändert wurde. Standardmäßig löst diese Funktion einfach einen Layoutprozeß aus.

Größenanpassung eines Gadgetfenster

Sie können für ein Gadgetfenster angeben, ob die Größe eines Gadget über die Funktion *SetShrinkWrap* angepaßt werden soll. Der Begriff Größenanpassung (Shrink Wrapping) besitzt für ein Gadgetfenster eine etwas andere Bedeutung als für ein Gadget selbst. Wird ein Gadgetfenster an eine Achse angepaßt, dann wird die Achsengröße auf der Grund-

lage der gewünschten Größen der Gadgets, die an dieser Achse angeordnet sind, automatisch berechnet.

Es ist möglich, die Größenanpassung in Abhängigkeit von der Breite und Höhe des Gadgetfensters ein- und auszuschalten:

```
void SetShrinkWrap(BOOL shrinkWrapWidth, BOOL shrinkWrapHeight);
```

Dabei:

- schaltet *shrinkWrapWidth* die horizontale Größenanpassung ein oder aus, je nach dem, ob TRUE oder FALSE übergeben wurde.
- schaltet *shrinkWrapHeight* die vertikale Größenanpassung ein oder aus, je nach dem, ob TRUE oder FALSE übergeben wurde.

Zugriff auf die Schriftart des Fensters

Mit Hilfe der Funktionen *GetFont* und *GetFontHeight* können Sie sich die aktuelle Schriftart und -größe ausgeben lassen:

```
TFont& GetFont();
UINT GetFontHeight() const;
```

Erfassen der Maus in einem Gadget

Wenn die linke Maustaste gedrückt wird, während sich der Mauszeiger im begrenzenden Rechteck eines Gadget befindet, wird dieses Gadget in jedem Fall benachrichtigt. Danach müssen Sie die Maus erfassen, wenn Sie Mausbewegungen weitersenden wollen. Dazu können Sie die Funktionen *GadgetsetCapture* und *GadgetReleaseCapture* verwenden:

```
BOOL GadgetsetCapture(TGadget& gadget);
void GadgetReleaseCapture(TGadget& gadget);
```

In beiden Funktionen gibt *gadget* an, für welches Gadget die Erfassung der Mausbewegungen aktiviert oder deaktiviert werden soll. Der von *GadgetSetCapture* zurückgegebene boolesche Wert stellt den Erfolg dar, ob das Erfassen gelungen ist.

Diese Funktionen werden gewöhnlich durch ein Gadget aufgerufen. Dieser Aufruf erfolgt über den Fenster-Zeiger des Gadget auf dessen Gadgetfenster.

Setzen des Hinweismodus

Der Hinweismodus eines Gadget schreibt vor, wann im übergeordneten Fenster des Gadgetfensters Hinweise zum Gadget angezeigt werden. Sie können den Hinweismodus eines Gadget über die Funktion *SetHintMode* setzen:

```
void SetHintMode(THintMode hintMode);
```

Das **enum** *THintMode* kann folgende drei Werte annehmen:

Hinweismodus-Flags

hintMode	Hinweisanzeige:
NoHints	Es werden keine Hinweise angezeigt.
PressHints	Hinweise werden angezeigt, wenn das Gadget angeklickt wird und solange, bis die Maustaste wieder freigegeben wird.
EnterHints	Hinweise werden angezeigt, wenn die Maus das Gadget passiert; das heißt, wenn die Maus in das Gadget eintritt.

Über die Funktion *GetHintMode* können Sie sich den aktuellen Hinweismodus ausgeben lassen:

```
THintMode GetHintMode();
```

Festgelegt wird der Hinweismodus über die Funktion *SetHintCommand*:

```
void SetHintCommand(int id);
```

Normalerweise wird diese Funktion von einem Gadget aus aufgerufen. Der Aufruf erfolgt über den *Fenster*zeiger auf das zugehörige Gadgetfenster. Die Funktion kann aber auch vom Gadgetfenster selbst aufgerufen werden. Im wesentlichen simuliert *SetHintCommand* eine Menüauswahl. Dadurch entspricht ein nachfolgendes Anklicken des Gadget dem Auswählen eines Menüpunktes.

Damit *SetHintCommand* auch ordnungsgemäß mit den Standardklassen von ObjectWindows funktioniert, müssen verschiedene Richtlinien beachtet werden:

- Das dem Gadgetfenster übergeordnete, dekorierte Rahmenfenster muß eine Meldungs- oder Statuszeile besitzen.
- Im Rahmenfenster müssen Hinweise enthalten sein.
- Es muß eine Stringressource geben, die die gleiche ID besitzt wie das Gadget, das heißt, wenn das Gadget die ID CM_MYGADGET hat, dann muß auch eine Stringressource als CM_MYGADGET definiert worden sein.

Behandlung von Idle-Aktionen

Für Gadgetfenster gibt es Standardroutinen für die Behandlung von Idle-Aktionen. Die Funktion *IdleAction* versucht, jedes im Fenster enthaltene Gadget zu aktivieren, indem die Funktion *CommandEnable* für alle Gadgets aufgerufen wird. Danach gibt die Funktion FALSE zurück.

```
BOOL IdleAction(long idleCount);
```

Durchsuchen von Gadgets

Über folgende Funktionen können die im Gadgetfenster eingebundenen Gadgets durchsucht werden:

```
TGadget* FirstGadget() const;
TGadget* NextGadget(TGadget& Gadget) const;
```

```
TGadget* GadgetFromPoint(TPoint& point) const;
TGadget* GadgetWithId(int id) const;
```

- *FirstGadget* liefert einen Zeiger auf das erste Gadget in der Gadgetliste eines Fensters.
- NextGadget liefert einen Zeiger auf das nächstfolgende Gadget in der Gadgetliste eines Fensters. Ist das aktuelle Gadget auch gleichzeitig das letzte Gadget des Fensters, dann gibt *NextGadget* den Wert 0 zurück.
- GadgetFromPoint liefert einen Zeiger auf das Gadget, auf das über *point* verwiesen wird. ist *point* kein Gadget, dann gibt *GadgetFromPoint* den Wert 0 zurück.
- GadgetWithId liefert einen Zeiger auf das Gadget, zu der die Gadget-ID *id* gehört. Besitzt kein Gadget des Fensters die angegebene Gadget-ID, dann gibt *GadgetWithId* den Wert 0 zurück.

Ableiten der Klasse TGadgetWindow

Über abgeleitete Klassen der Klasse *TGadgetWindow* können Sie sich eigene, spezielle Gadgetfenster erzeugen. *TGadgetWindow* enthält verschiedene **protected**-Zugriffsfunktionen, mit deren Hilfe Sie eine Gadgetklasse von *TGadgetWindow* ableiten können.

Zeichnen eines Gadgetfenster

Wie bei regulären Fenster ist auch in der Klasse *TGadgetWindow* die Funktion *Paint* implementiert:

```
void Paint(TDC& dc, BOOL erase, TRect& rect);
```

Durch die Implementierung dieser Funktion wird die Schriftart des Fensters im Gerätekontext markiert und die Funktion *PaintGadgets* aufgerufen:

```
virtual void PaintGadgets(TDC& dc, BOOL erase, TRect& rect);
```

Durch *PaintGadgets* wird jedes Gadget innerhalb des Fensters zur Bildschirmausgabe aufgefordert. Überschreiben Sie *PaintGadgets,* um für Ihre Fenster ein benutzerdefiniertes Aussehen zu implementieren, etwa für Trennlinien, Hervorhebung, usw.

Größe und inneres Rechteck

Mit Hilfe der Funktionen *GetDesiredSize* und *GetInnerRect* können Sie die erforderliche Gesamtgröße (das ist die Größe, die für die Unterbringung von Rahmen, Seitenrändern sowie des breitesten oder höchsten Gadgets erforderlich ist) und die Größe bzw. die Position des inneren Rechtecks im Fenster ermitteln.

```
virtual void GetDesiredSize(TSize& size);
```

Wurde für das Fenster Größenanpassung aktiviert, dann berechnet *GetDesiredSize* die für das Fenster erforderliche Größe, um Rahmen, Seitenränder sowie das breiteste oder höchste Gadget unterzubringen. Ist das nicht der Fall, dann verwendet die Funktion *GetDesiredSize* die aktuellen Werte für Breite und Höhe. Danach wird das Ergebnis in *size* abgelegt.

```
virtual void GetInnerRect(TRect& rect);
```

Die Funktion *GetInnerRect* berechnet den Bereich innerhalb des Rahmens und der Seitenränder des Fensters. Das Ergebnis wird in *rect* abgelegt.

Sie können *GetDesiredSize* und *GetInnerRect* so überschreiben, daß genügend Platz für ein benutzerspezifisches Aussehen Ihres Fensters verbleibt. Sie sollten jedoch beachten, daß Sie auch die andere Funktion überschreiben müssen, wenn Sie eine dieser beiden Funktionen überschrieben haben.

Layouteinheiten

In einem Gadgetfenster können drei verschiedene Maßeinheiten verwendet werden:

- Pixel beruhen auf einem einzelnen Bildschirmpixel.
- Layout-Einheiten sind logische Einheiten, die über die Teilung der Fensterschrift "em" in 8 vertikale und 8 horizontale Segmente definiert sind.
- Rahmeneinheiten basieren auf der Dicke eines Fensterrahmens. Dies entspricht gewöhnlich einem Pixel, bei höheren Bildschirmauflösungen kann dieser Wert jedoch auch größer sein.

Normalerweise erweist es sich als besonders günstig, mit Layouteinheiten zu arbeiten. Da sie auf der Schriftgröße beruhen, brauchen Sie sich nicht um die Skalierung Ihrer Einheiten zu kümmern, wenn Sie die Fenstergröße oder die Systemmetrik ändern.

Ist es einmal erforderlich, die Layouteinheiten in Pixel umzuwandeln, dann können Sie für die Lösung dieser Aufgabe die Funktion *LayoutUnitsToPixels* verwenden:

```
int LayoutUnitsToPixels(int units);
```

Dabei stellt *units* das Maß für die Layouteinheiten dar, die in Pixel konvertiert werden sollen. *LayoutUnitsToPixels* liefert die Anzahl der Pixel zurück, die *units* Layouteinheiten entsprechen.

Über die Funktion *GetMargins* können Sie auch ein *TMargins*-Objekt in Pixeleinheiten umwandeln:

```
void GetMargins(TMargins& margins,
                int& left,
                int& right,
                int& top,
                int& bottom);
```

Dabei ist:

- *margins* ein Objekt mit den Maßen, die umgewandelt werden sollen. Das können Angaben in Pixel, Layouteinheiten oder Rahmeneinheiten sein.
- In *left, right, top* und *bottom* werden die Ergebnisse der Umwandlung abgelegt.

Antwortfunktionen von Botschaften

In der Klasse *TGadgetWindow* treten folgende Ereignisse auf:

- WM_CTLCOLOR
- WM_LBUTTONDOWN
- WM_LBUTTONUP
- WM_MOUSEMOVE
- WM_SIZE
- WM_SYSCOLORCHANGE

Die entsprechenden Ereignisbehandlungsfunktionen werden von *TGadgetWindow* ebenfalls implementiert.

Gadgetfensterklassen in ObjectWindows

ObjectWindows beinhaltet viele verschiedene Klassen, die von der Klasse *TGadgetWindow* abgeleitet wurden. Diese Fenster stellen verschiedene Möglichkeiten für das Anzeigen und Anordnen von Gadgets dar. Folgende Gadgetfensterklassen sind in ObjectWindows enthalten:

- *TControlBar*
- TMessageBar
- TStatusBar
- TToolBox

In den nächsten Abschnitten werden die oben aufgeführten Klassen erläutert.

Die Klasse TControlBar

Die Klasse *TControlBar* implementiert einen Steuerbalken, der einem "Tool-Balken" oder einem "Steuerbalken" ähnelt, wie man sie am oberen Fensterrand vieler bekannter Anwendungen findet. In einem Steuerbalken können Sie alle Gadgettypen plazieren.

Der Konstruktor für *TControlBar* besitzt folgendes Aussehen:

```
TControlBar(TWindow* parent = 0,
            TTileDirection direction = Horizontal,
            TFont* font = new TGadgetWindowFont,
            TModule* module = 0);
```

wobei:

- das Argument *parent* ein Zeiger auf das übergeordnete Fenster des Steuerbalkens enthält.
- direction ein **enum** *TTileDirection* darstellt. *direction* kann einer der beiden folgenden Werte annehmen: *Horizontal* oder *Vertical*.
- font ein Zeiger auf ein *TFont*-Objekt ist. Dieses Objekt enthält die Schriftart für das Gadgetfenster. Standardmäßig ist es auf *TGadgetWin-*

dowFont gesetzt, also auf eine Sans-Serif-Schriftart mit variabler Größe, normalerweise Helvetica.

- module wird an den *TWindow*-Basiskonstruktor als *TModule*-Parameter übergeben. Dieser Parameter besitzt die Voreinstellung 0.

Die Klasse TMessageBar

Die Klasse *TMessageBar* implementiert eine Meldungszeile ohne Rahmen sowie ein Textgadget von der Breite des Fensters selbst. Es positioniert sich selbst horizontal entlang des unteren Randes des übergeordneten Fensters.

Erzeugen und Löschen einer Instanz der Klasse TMessageBar

Der Konstruktor für die Klasse *TMessageBar* besitzt folgende Syntax:

```
TMessageBar(TWindow*   parent = 0,
            TFont*     font = new TGadgetWindowFont,
            TModule*   module = 0);
```

wobei:

- *parent* ein Zeiger auf das übergeordneten Fensters des Steuerbalken ist.
- font einen Zeiger auf ein *TFont*-Objekt ist. Dieses Objekt enthält die Schriftart für das Gadgetfenster. Standardmäßig ist es auf *TGadgetWindowFont* gesetzt, das heißt, auf eine Sans-Serif-Schriftart mit variabler Größe, normalerweise Helvetica.
- module wird an den *TWindow*-Basiskonstruktor als *TModule*-Parameter übergeben. Dieser Parameter besitzt die Voreinstellung 0.

Die Funktion *~TMessageBar* gibt den Textspeicher des Objektes wieder frei.

Angaben von Text für die Meldungszeile

Mit Hilfe der Funktion *SetText* können Sie den Text für die Meldungszeile eines Textgadget angeben:

```
void SetText(const char* text);
```

Über diese Funktion wird der String t*ext* in der Meldungszeile ausgegeben.

Hinweistext angeben

Über die Funktion *SetHintText* können Hinweistexte zu den einzelnen Menüs oder Menüpunkten vorgegeben werden, die in einem eigenen Feld über der Meldungszeile ausgegeben werden:

```
virtual void SetHintText(const char* text);
```

Wenn Sie für t*ext* eine 0 angeben, wird der Hinweistext gelöscht.

Die Klasse TStatusBar

Die Klasse *TStatusBar* ähnelt der Klasse *TMessageBar*. Der Unterschied zwischen beiden besteht darin, daß Statuszeilen mehr Optionen besitzen als eine einfache Meldungszeile. Dies sind zum Beispiel Mehrfachtexttgadgets und der für die Anzeige von Tastaturmodi, freigehaltene Bereich (wie *CapsLock*, *Insert* oder *Overwrite*).

Erzeugen und Löschen einer Instanz der Klasse TStatusBar

Der Konstruktor für die Klasse *TStatusBar* besitzt folgendes Aussehen:

```
TStatusBar(TWindow* parent = 0,
           TGadget::TBorderStyle borderStyle = TGadget::Recessed,
           UINT modeIndicators = 0,
           TFont* font = new TGadgetWindowFont,
           TModule* module = 0);
```

wobei:

- *parent* einen Zeiger auf das übergeordnete Fensterobjekt darstellt.
- style ein **enum** *TBorderStyle* ist.
- modeIndicators angibt, welcher Tastaturmodus in der Statuszeile angezeigt werden kann. Ein als **enum-**Typ definierter *TModeIndicator* liefert für diesen Parameter folgende gültigen Werte: *ExtendSelection*, *CapsLock*, *NumLock*, *ScrollLock*, *Overtype* und *RecordingMacro*.

Es können mehrere dieser Werte gleichzeitig als ORed angezeigt werden, um die Anzeige von Mehrfachtastaturmodi anzustellen.

- *font* ein Zeiger auf ein *TFont*-Objekt ist. Dieses Objekt enthält die Schriftart für das Gadgetfenster.
- module wird an den *TWindow*-Basiskonstruktor als *TModule*-Parameter übergeben. Dieser Parameter besitzt die Voreinstellung 0.

Einfügen von Gadgets in eine Statuszeile

TStatusBar überschreibt die Standardfunktion *Insert*. Standardmäßig fügt die Funktion *TStatusBar* das neue Gadget nach den bereits existierenden Textgadgets aber vor den Modusanzeige-Gadgets ein.

Sie können ein Gadget innerhalb der Statuszeile neben ein schon vorhandenes Gadget setzen, indem Sie einen Zeiger auf dieses existierende Gadget an die Funktion *Insert* übergeben, das dem neuen Gadget gleichgeordnet sein muß. Hinter den Modusanzeigen dürfen jedoch keine Gadgets eingefügt werden.

Anzeigen von Modusindikatoren

Soll eine bestimmte Modusanzeige in der Statuszeile erscheinen, müssen Sie diesen Modus während der Erzeugung der Statuszeile angeben. Ein Modus wird solange über die Statuszeile angezeigt, bis Sie die Anzeige verändern. Die Klasse *TStatusBar* enthält verschiedene Funktionen für die Anordnung der Modusanzeige.

Mit Hilfe der Funktion *SetModeIndicator* können Sie eine Modusanzeige in jeden anderen gültigen Status umschalten:

```
void SetModeIndicator(TModeIndicator indicator, BOOL state);
```

wobei:

- *indicator* die Modusanzeige darstellt, die festgelegt werden soll. Hierbei kann jeder beliebige Wert des im Konstruktor verwendeten **enum** *TModeIndicator* eingegeben werden.
- state der Status ist, in den die Modusanzeige versetzt werden soll.

Über die Funktion *ToggleModeIndicator* können Sie eine Modusanzeige auch hin- und herschalten:

```
void ToggleModeIndicator(TModeIndicator indicator);
```

Dabei ist *indicator* die Modusanzeige, die umgeschaltet werden soll. Das Argument kann jeden beliebigen Wert des im Konstruktor verwendeten **enum** *TModeIndicator* annehmen.

Anordnen von Gadgets in der Statuszeile

Mit Hilfe der Funktion *SetSpacing* können Sie in der Statuszeile den Raum zwischen den Gadgets für die Modusanzeige variieren:

```
void SetSpacing(TSpacing& spacing);
```

Das Argument *spacing* stellt hierbei eine Referenz auf ein *TSpacing*-Objekt dar. *TSpacing* ist eine Struktur, die in der Klasse *TStatusBar* definiert ist. Sie enthält zwei Datenelemente: ein *TMargins::TUnits*-Element mit der Bezeichnung *Units* und ein Integerelement mit der Bezeichnung *Value*. Über den Konstruktor *TSpacing* wird *Units* auf *TMargins::LayoutUnits* und *Value* auf 0 gesetzt.

Mit Hilfe der Struktur *TSpacing* können Sie die Maßeinheit und die Anzahl der Einheiten in einem einzigen Objekt angeben. Wenn Sie dieses Objekt an den Befehl *SetSpacing* übergeben, dann wird der Zwischenraum zwischen den einzelnen Gadgets für die Modusanzeige auf *Value Units* gesetzt. Damit sich die Änderungen auswirken, müssen Sie die Statuszeile neu ordnen.

Die Klasse TToolBox

Die Klasse *TToolBox* unterscheidet sich von den anderen bisher besprochenen Klassen von Gadgetfenster in ObjectWindows dadurch, daß die in diesem Fenster enthaltenen Gadgets nicht in einer einzelnen Reihe aufgelistet werden. Statt dessen werden die Gadgets in einer Matrix angeordnet. Alle Spalten dieser Matrix besitzen die gleiche Breite (sie sind so breit wie das breiteste Gadget) und alle Zeilen der Matrix sind gleich hoch (so hoch wie das höchste Gadget). Die Gadgets werden so angeordnet, daß sich die Rahmen überlappen und vom Rahmen der Toolbox verdeckt werden.

Um den Palettentyp Toolbox zu erstellen, kann *TToolBox* als untergeordnetes Fenster in einem *TFloatingFrame* erzeugt werden. Als Erläuterung hierfür dient das Programmbeispiel PAINT, das sich im Verzeichnis EXAMPLES\OWL\OWLAPPS\PAINT befindet.

Erzeugen und Löschen einer Instanz der Klasse TToolBox

Der Konstruktor der Klasse *TToolBox* besitzt folgende Syntax:

```
TToolBox(TWindow* parent,
         int numColumns = 2,
         int numRows = AS_MANY_AS_NEEDED,
         TileDirection direction = Horizontal,
         TModule* module = 0);
```

wobei:

- *parent* einen Zeiger auf das übergeordnete Fensterobjekt darstellt.
- numColumns die Anzahl der Spalten in der Toolbox angibt.
- numRows die Anzahl der Zeilen in der Toolbox angibt.
- direction ein **enum** *TTileDirection* ist. Das Argument *direction* kann einen der beiden folgenden Werte annehmen: *Horizontal* oder *Vertical*. Ist *direction Horizontal*, dann werden die Gadgets von links nach rechts nebeneinander angeordnet. Dabei wird an der linken oberen Ecke begonnen und jeweils eine Zeile nach unten gesprungen, wenn die aktuelle Zeile voll ist. Ist *direction Vertical*, dann werden die Gadgets von oben nach unten nebeneinander angeordnet. Dabei wird an der linken oberen Ecke begonnen und jeweils eine Spalte nach rechts gesprungen, wenn die aktuelle Spalte voll ist.
- module wird an den Basiskonstruktor *TWindow* als *TModule*-Parameter übergeben. Dieser Parameter besitzt die Voreinstellung 0.

Die Konstante AS_MANY_AS_NEEDED können Sie entweder für *numColumns* oder für *numRows* angeben, nicht aber für beide gleichzeitig. Haben Sie diese Konstante für einen der beiden Parameter übergeben, berechnet die Toolboxfunktion, wieviele Teilungen auf der Basis der gegensätzlichen Dimensionen erforderlich sind. Wenn Sie beispielsweise 20 Gadgets haben und Sie fordern, daß vier Spalten angelegt werden, dann bekommen Sie 5 Zeilen.

Wechseln von Toolbox-Dimensionen

Mit Hilfe der Funktion *SetDirection* können Sie zwischen den Dimensionen der Toolbox hin- und herschalten:

```
virtual void SetDirection(TTileDirection direction);
```

Dabei enthält *direction* ein **enum** *TTileDirection*. Dieses Argument kann einen der beiden folgenden Werte annehmen: *Horizontal* oder *Vertical*.

Entspricht der Wert von *direction* nicht der aktuellen Richtung für die Toolbox, dann schaltet die Toolbox ihren Zeilen- und Spaltenzähler um. Angenommen, Ihre Toolbox besäße drei Spalten und fünf Zeilen und ist vertikal angeordnet. Wenn Sie nun die Funktion *SetDirection* aufrufen und dabei *direction* auf *Horizontal* setzen, dann verändert die Toolbox ihre Anordnung derart, daß sie danach fünf Spalten und drei Zeilen enthält.

K A P I T E L 14

Druckerobjekte

Dieses Kapitel beschreibt Klassen von ObjectWindows, die Ihnen bei den folgenden Aufgaben im Zusammenhang mit dem Drucken helfen. Zwei Klassen von ObjectWindows erleichtern diese Aufgaben:

Siehe die Online-Datei OWLDOC.WRI für Informationen über die Klassen zur Druckvoranzeige.

- *TPrinter* kapselt das Druckerverhalten und den Zugriff zu den Druckertreibern. Sie ist zur Konfiguration des Druckes fähig, indem sie einen Dialog anzeigt. Dieser Dialog erlaubt dem Benutzer die Auswahl des gewünschten Druckers wie auch die Festlegung der aktuellen Druckereinstellungen, so etwa die grafische Auflösung und das Längs- oder Querformat.
- *TPrintout* kapselt den aktuellen Ausdruck. Ihre Verhältnis zum Drucker ist mit dem Verhältnis von *TWindow* zum Bildschirm vergleichbar. Das Zeichnen auf den Bildschirm erfolgt in der Elementfunktion *Paint* des *TWindow*-Objektes. Dem gegenüber erfolgt das Schreiben auf dem Drucker in der Elementfunktion *PrintPage* des *TPrintout*-Objektes. Um etwas auf dem Drucker auszudrucken, übergibt die Anwendung eine Instanz von *TPrintout* an eine Instanz der Elementfunktion *Print* von *TPrinter*.

Dieses Kapitel beschreibt die folgenden Druckaufgaben:

- Erzeugen eines Druckerobjektes
- Erzeugen eines Ausdrucksobjektes
- Drucken von Fensterinhalten
- Drucken eines Dokumentes
- Senden des Ausdrucks zum Drucker
- Auswählen und Konfigurieren eines Druckers

Erzeugen eines Druckerobjektes

Der einfachste Weg, ein Druckerobjekt zu erzeugen, ist die Deklaration eines *TPrinter** innerhalb eines Fensterobjektes, das andere Objekte im Programm für ihren Druckbedarf nutzen.

```
class MyWindow: public TFrameWindow {
  TPrinter* Printer;

//...

};
```

Um einen Drucker verfügbar zu machen, lassen Sie *Printer* auf eine Instanz von *TPrinter* zeigen. Dies kann im Konstruktor geschehen:

```
MyWindow::MyWindow(TWindow* parent, char *title){

//...

Printer = new TPrinter;
}
```

Ebenso sollten Sie im Destruktor das Druckerobjekt eliminieren:

```
MyWindow::~MyWindow() {

//...

  delete Printer
}
```

Hier ist die Lösung aus dem Beispiel PRINTING.CPP im Verzeichnis OWLAPI\PRINTING:

```
class TRulerWin : public TFrameWindow {
  TPrinter* Printer;

//...

}

TRulerWin::TRulerWin(TWindow* parent, const char* title, TModule* module)
  : TFrameWindow(parent, title, 0, FALSE, module),
    TWindow(parent, title, module)
{

//...

  Printer = new TPrinter;
}
```

Bei den meisten Anwendungen reicht dies aus. Das Hauptfenster der Anwendung initialisiert ein Druckerobjekt, welches den in WIN.INI angegebenen Standarddrucker verwendet. In manchen Fällen können Anwendungen aber auch verschiedene Drucker von verschiedenen Fenstern aus gleichzeitig benutzen. Konstruieren Sie in diesem Fall ein Druckerobjekt

im Konstruktor von jedem entsprechenden Fenster und wechseln Sie dann den Druckertreiber für einen ober mehrere Drucker. Wenn das Programm verschiedene Drucker zu verschiedenen Zeiten benutzt, wird es wahrscheinlich besser sein, dasselbe Druckerobjekt zu nutzen und die verschiedenen Drucker nach Bedarf auszuwählen.

Auch wenn Sie versucht sind, den Konstruktor von *TPrinter* zu überschreiben, um einen anderen Drucker als den der Systemvoreinstellung zu nutzen, ist es eher empfehlenswert, immer den Standardkonstruktor zu verwenden und dann das mit dem Objekt assoziierte Gerät zu ändern.

Erzeugen eines Ausdrucksobjektes

Grafikfunktionen von Windows sind in Kapitel 15 erklärt.

Das Erzeugen eines Ausdrucksobjektes ist mit dem Schreiben einer Elementfunktion *Paint* für ein Fensterobjekt vergleichbar: Sie verwenden Grafikfunktionen von Windows, um ein Bild Ihrer Wahl auf dem Gerätekontext zu generieren. Der Bildschirmkontext des Fensterobjektes übernimmt die Interaktionen mit dem Bildschirm; der Gerätekontext des Ausdrucksobjektes versteckt vor Ihnen das Druckergerät auf fast gleiche Art.

Um ein Ausdrucksobjekt zu erzeugen, gehen Sie wie folgt vor:

- Leiten Sie einen neuen Objekttyp von *TPrintout* ab, der die Elementfunktion *PrintPage* überschreibt. In sehr einfachen Fällen sind Sie damit schon fertig.
- Wenn das Dokument mehr als eine Seite besitzt, müssen Sie auch die Elementfunktion *HasPage* überschreiben. Sie muß einen Wert ungleich Null zurückgeben. solange noch eine andere Seite gedruckt werden muß. Die aktuelle Seitennummer wird als Parameter an *PrintPage* übergeben.

Siehe das *ObjectWindows Referenzhandbuch* für eine Beschreibung der Klasse *TPrintout*.

Das Ausdrucksobjekt besitzt Felder, die die Größe der Seite und einen Gerätekontext enthalten, der bereits für die Übergabe an den Drucker initialisiert ist. Das Druckerobjekt setzt diese Werte über den Aufruf der Elementfunktion *SetPrintParams* des Ausdrucksobjektes. Sie sollten den Gerätekontext des Ausdrucksobjektes in allen Aufrufen von Grafikfunktionen von Windows verwenden.

Hier ist die von *TPrintout* abgeleitete Klasse *WindowPrintout* aus dem Beispiel PRINTING.CPP:

```
class TWindowPrintout : public TPrintout {
  public:
    TWindowPrintout(const char* title, TWindow* window);
```

```
    void GetDialogInfo(int& minPage, int& maxPage,
                       int& selFromPage, int& selToPage);
    void PrintPage(int page, TRect& rect, unsigned flags);
    void SetBanding(BOOL b) {Banding = b;}
    BOOL HasPage(int pageNumber) {return pageNumber == 1;}

  protected:
    TWindow* Window;
    BOOL     Scale;
};
```

GetDialogInfo erhält von einem Dialogfenster Information über den Bereich an Seiten, sofern die Auswahl von Seiten möglich ist. Da es nur eine Seite gibt, sieht *GetDialogInfo* für *TWindowPrinout* wie folgt aus:

```
TWindowPrintout::GetDialogInfo(int& minPage, int& maxPage,
                               int& selFromPage, int& selToPage)
{
  minPage = 0;
  maxPage = 0;
  selFromPage = selToPage = 0;
}
```

PrintPage muß überschrieben werden, um den Inhalt von jeder Seite, jedem Abschnitt (falls abschnittweise Bearbeitung eingeschaltet ist) oder Fenster zu drucken. *PrintPage* sieht für *TWindowPrintout* wie folgt aus:

```
void TWindowPrintout::PrintPage(int, TRect& rect, unsigned)
{
  // Skaliere eventuell den DC im Verhältnis zum Fenster,
  // damit der Ausdruck dem Fenster gleicht.
  //
  int    prevMode;
  TSize  oldVExt, oldWExt;
  if (Scale) {

    prevMode = DC->SetMapMode(MM_ISOTROPIC);
    TRect windowSize = Window->GetClientRect();
    DC->SetViewportExt(PageSize, &oldVExt);
    DC->SetWindowExt(windowSize.Size(), &oldWExt);
    DC->IntersectClipRect(windowSize);
    DC->DPtoLP(rect, 2);
  }

  // Veranlassen Sie, daß das Fenster sich selbst darstellt
  Window->Paint(*DC, FALSE, rect);

  // Stellen Sie die im DC vorgenommenen Änderungen wieder her
  if (Scale) {
    DC->SetWindowExt(oldWExt);
    DC->SetViewportExt(oldVExt);
    DC->SetMapMode(prevMode);
```

```
  }
}
```

SetBanding wird mit eingeschalteter abschnittweiser Bearbeitung aufgerufen:

```
printout.SetBanding(TRUE);
```

HasPage wird nach jeder gedruckten Seite aufgerufen und gibt standardmäßig FALSE zurück, das bedeutet, daß nur eine Seite gedruckt wird. Diese Funktion muß überschrieben werden, damit sie den Wert TRUE zurückgibt, solange Seiten in einem mehrseitigen Dokument übrig sind.

Drucken von Fensterinhalten

Die einfachste Art, einen Ausdruck zu generieren, ist die Kopie eines Fensters, da Fenster nur eine Seite besitzen und ein Fensterobjekt bereits einen Weg kennt, sich selbst auf einem Gerätekontext darzustellen.

Das Erzeugen eines Fenster-Ausdrucksobjektes erfordert nur einen einzigen Schritt. Sie müssen nur ein Fenster-Ausdrucksobjekt konstruieren und an das Objekt einen Titel-String sowie einen Zeiger auf des zu druckende Fenster übergeben:

```
TWindowPrintout printout("Ruler Test", this);
```

Oftmals soll ein Fenster einen Ausdruck von sich selbst in Antwort auf ein Menükommando erzeugen. Hier ist die Elementfunktion zur Beantwortung von Botschaften in PRINTING.CPP, die auf das Druckkommando antwortet:

```
void TRulerWin::CmFilePrint()      // Führe das Kommando File:Print aus
{
  if (Printer) {
    TWindowPrintout printout("Ruler Test", this);
    printout.SetBanding(TRUE);
    Printer->Print(this, printout, TRUE);
  }
}
```

Diese Elementfunktion ruft die Elementfunktion *Print* des Druckerobjektes auf, die einen Zeiger auf das übergeordnete Fenster und einen Zeiger auf das Ausdrucksobjekt übergibt. Ferner gibt sie an, ob das Druckerdialogfenster angezeigt werden soll oder nicht.

TWindowPrintout druckt sich selbst aus, indem sie die Elementfunktion *Paint* Ihres Fensterobjektes innerhalb von *TWindowPrintout::PrintPage* mit einem Druckergerätekontext an Stelle eines Bildschirmkontextes aufruft.

Drucken eines Dokumentes

Windows sieht einen Ausdruck als eine Serie von Seiten an. Deshalb muß Ihr Ausdrucksobjekt für einen Ausdruck unter Windows ein Dokument in eine Serie von Seitenbildern umwandeln. Genauso wie Sie Fensterobjekte für das Darstellen von Bildern auf dem Bildschirm in Windows verwenden, benutzen Sie auch Ausdrucksobjekte, um Bilder auf dem Drucker auszugeben.

Ihr Ausdrucksobjekt muß dabei zu drei Dingen in der Lage sein:

- Setzen der Druckparameter
- Zählen seiner Seiten
- Zeichnen jeder Seite auf einen Gerätekontext
- Anzeigen, ob es noch mehr Seigen gibt

Die folgenden Abschnitte beschreiben jede dieser Fähigkeiten.

Setzen der Druckparameter

Bevor Ihr Dokument aufgefordert wird, sich selbst zu drucken, gibt das Druckerobjekt (abgeleitet von der Klasse *TPrinter*) Ihrem Dokument die Chance, sich selbst in Seiten aufzuteilen. Dazu werden die zwei Elementfunktionen *SetPrintParams* und dann *GetDialogInfo* des Ausdrucksobjektes aufgerufen.

Die Funktion *SetPrintParams* initialisiert die Seitengröße und Variablen des Gerätekontextes im Ausdrucksobjekt. *SetPrintParams* dient ebenso zur Berechnung aller für einen effizienten Ausdruck einzelner Seiten eventuell benötigten Informationen. *SetPrintParams* kann zum Beispiel berechnen, wieviele Textzeilen in der ausgewählten Schriftart in den Druckbereich passen (mit Hilfe der Windows-API *GetTextMetrics*).

Wenn Sie *SetPrintParams* überschreiben, sollten Sie sicher stellen, daß die geerbte Elementfunktion aufgerufen wird, die den Gerätekontext des Ausdrucksobjektes und die Standardseitengrößen setzt.

Zählen von Seiten

Nach dem Aufruf von *SetPrintParams* ruft das Druckerobjekt *GetDialogInfo* auf. *GetDialogInfo* erhält vom Druckerdialogfenster die vom Benutzer angegebene Seitenbereichsinformation. Die Funktion kann auch zum Berechnen der Gesamtzahl von Seiten dienen, wobei sie auf die in *SetPrintParams* ermittelten Seitengröße zurückgreift.

Drucken jeder Seite

Wenn ein Druckerobjekt dem Dokument die Möglichkeit gegeben hat, sich selbst in Seiten aufzuteilen, ruft es die Elementfunktion *PrintPage* des Ausdrucksobjektes für jede zu druckende Seite auf. Der Prozeß, genau den Teil des Dokumentes auszudrucken, der zur gegebenen Seite gehört, ähnelt der Entscheidung, welcher Bereich in einem bildlauffähigen

Fenster dargestellt wird. Dabei gibt es zwei Punkte, die Sie im Auge behalten sollten, wenn sie *PrintPage*-Elementfunktionen schreiben:

- Geräteunabhängigkeit. Stellen Sie sicher, daß Ihr Code keine Annahmen über Skalierung, XY-Verhältnis oder Farben voraussetzt. Solche Eigenschaften können bei den verschiedenen Video- und Druckergeräten variieren. Deshalb sollten Sie besser alle Geräteabhängigkeiten aus Ihrem Code entfernen.
- Fähigkeiten des Gerätes. Zwar unterstützen die meisten Videogeräte alle GDI-Operationen, bei einigen Druckern ist dies allerdings nicht der Fall. So akzeptieren beispielsweise viele Druckergeräte wie Plotter überhaupt keine Bitmaps. Andere unterstützen nur bestimmte Operationen. Bei komplexen Druckaufgaben sollten Sie die Windows-API-Funktion *GetDeviceCaps* aufrufen, die wichtige Informationen über die Fähigkeiten des gegebenen Ausgabegerätes zurückliefert.

Anzeigen weiterer Seiten

Ausdrucksobjekte haben eine letzte Aufgabe: sie zeigen dem Druckerobjekt an, ob nach der gegebenen Seite noch druckbare Seiten vorliegen. Die Elementfunktion *HasPage* erwartet eine Seitennummer als Parameter und gibt ein Boolean-Wert zurück, der noch weitere existierende Seiten anzeigt. *HasPage* gibt standardmäßig TRUE nur für die erste Seite zurück. Um mehrere Seiten zu drucken, muß Ihre Ausdrucksobjekt *HasPage* überschreiben, damit TRUE zurückgegeben wird, wenn das Dokument weitere zu druckende Seiten besitzt, und FALSE, wenn der übergebene Parameter die letzte Seite ist.

Stellen Sid sicher, daß *HasPage* an irgendeinem Punkt FALSE zurückliefert. Wenn *HasPage* immer TRUE zurückgibt, geht das Ausdrucken in eine Endlosschleife.

Weitere Betrachtungen zum Ausdrucken

Ausdrucksobjekte besitzen mehrere andere Elementfunktionen, die Sie bei Bedarf überschreiben können. *BeginPrinting* und *EndPrinting* werden vor bzw. nach dem Ausdrucken eines jeden Dokumentes aufgerufen. Wenn Sie speziellen Code für weitere Einstellungen benötigen, können Sie ihn in *BeginPrinting* einbauen und diese Einstellungen in *EndPrinting* wieder zurücknehmen.

Der Ausdruck von Seiten erfolgt sequenziell. Das heißt, daß der Drucker für jede Seite in einer Folge *PrintPage* aufruft. Vor dem ersten Aufruf von *PrintPage*, ruft das Druckerobjekt in jedem Fall *BeginDocument* auf und übergibt die Nummern der ersten und letzten zu druckenden Seiten. Wenn sich Ihr Dokument bei einer anderen Seite als der ersten auf den Beginn des Ausdruckes vorbereiten muß, dann überschreiben Sie *BeginDocument*. Die korrespondierende Elementfunktion *EndDocument* wird nach dem Ausdruck der letzten Seite aufgerufen.

Wenn mehrere Kopien ausgedruckt werden, können mehrere Paare von *BeginDocument*/*EndDocument* zwischen *BeginPrinting* und *EndPrinting* aufgerufen werden.

Auswählen verschiedener Drucker

Wenn Sie ein Druckerobjekt in Ihrer Anwendung haben, können Sie es mit jedem, in Windows installierten Druckergerät assoziieren. Standardmäßig benutzt *TPrinter* den Standarddrucker von Windows, wie er im [devices]-Abschnitt der Datei WIN.INI angegeben ist.

Es gibt zwei Möglichkeiten, einen alternativen Drucker anzugeben: direkt im Code und über ein Dialogfenster für den Benutzer.

Auswählen eines Druckers

Der am ehesten übliche Weg, einen anderen Drucker zuzuweisen, besteht in der Anzeige eines Dialogfensters, damit der Benutzer aus einer Liste mit installierten Druckergeräten aussuchen kann. *TPrinter* führt dies automatisch durch, wenn Sie deren Elementfunktion *Setup* aufrufen. *Setup* zeigt ein auf *TPrinterDialog* basierendes Dialogfenster an.

Konfigurieren des Druckers

Einer der Schalter im Druckerdialog erlaubt dem Benutzer die Änderung der Druckerkonfiguration. Der Schalter »Einrichten« öffnet ein Konfigurationsdialogfenster, das im Gerätetreiber des Druckers definiert ist. Ihrer Anwendung hat keinerlei Kontrolle über das Aussehen und die Funktion der Konfigurationsdialogfensters aus dem Treiber.

Zuweisen eines speziellen Druckers

In manchen Fällen könnte die Zuweisung eines speziellen Druckergerätes an Ihr Druckerobjekt ohne Benutzereingabe nötig sein. *TPrinter* besitzt eine Elementfunktion *SetPrinter*, die genau dies erledigt.

SetPrinter erwartet drei Strings als Parameter: einen Gerätenamen, einen Treibernamen und ein Port-Namen.

K A P I T E L 15

Grafikobjekte

Dieses Kapitel beschreibt die ObjectWindows 2.5 Kapselung der Windows-GDI. ObjectWindows 2.5 erleichtert die Verwendung der GDI-Grafikobjekte und -funktionen durch die Vereinfachung der Erzeugung und Manipulation von GDI-Objekten. Von einfachen Objekten wie Stift und Pinsel bis hin zu komplexeren Objekten wie Schriftarten und Bitmaps bietet die GDI-Kapselung der ObjectWindows-Bibliothek ein einfaches, konsistentes Modell für grafische Programmierung unter Windows.

GDI-Klassenorganisation

Es gibt eine Reihe von ObjectWindows-Klassen für die Kapselung der GDI-Funktionalität. Die meisten sind dabei von der Klasse *TGdiObject* abgeleitet. *TGdiObject* stellt eine allgemeine Funktionalität für alle ObjectWindows-GDI-Klassen bereit.

TGdiObject ist die abstrakte Basisklasse für ObjectWindows-GDI-Objekte. Sie stellt einen Basisdestruktor, einen HGDIOBJ-Konvertierungoperator und die Basisfunktion *GetObject* bereit. Sie bietet ferner für echte GDI-Objekte (d.h. für Objekte, die von *TGdiObject* abgeleitet sind. Andere GDI-Objekte wie zum Beispiel *TRegion*, *TIcon* und *TDib*, sind von *TGdiBase* abgeleitet und werden deshalb auch *pseudo-GDI-Objekte* genannt) einen Kontrollmechanismus für verwaiste Objekte (das sind Objekte, auf die keine Referenz mehr besteht).

Die anderen Klassen in der ObjectWindows-GDI-Kapselung sind:

- *TDC* ist die Stammklasse für die Kapselung der ObjectWindows-GDI-Gerätekontexte. Sie können ein *TDC*-Objekt direkt erzeugen oder abgeleitete Klassen für spezialisierteres Verhalten verwenden.
- TPen enthält die Funktionalität von Stiftobjekten von Windows. Sie können ein Stiftobjekt ohne Vorgabe, von einem existierenden Stift-Handle, einem Objekt oder einer logischen Stiftstruktur (LOGPEN) aus konstruieren.

- TBrush enthält die Funktionalität von Windows Stiftobjekten. Sie können einen selbstdefinierten soliden, stilisierten oder gemusterten Pinsel konstruieren, Sie können ein existierendes Pinsel-Handle oder Pinselobjekt verwenden oder Sie gehen von einer logischen Pinselstruktur (LOGBRUSH) aus.
- TFont ermöglicht eine einfache Verwendung der Windows-Schriften. Sie können eine Schrift mit eigenen Spezifikationen, von einem existierenden Schrift-Handle, einem Objekt oder einer logischen Schriftstruktur (LOGFONT) aus konstruieren.
- TPalette kapselt eine GDI-Palette. Sie können eine neue Palette konstruieren oder existierende Paletten von verschiedenen Farbtabellentypen, die von DIBs verwendet werden, benutzen.
- TBitmap enthält Windows-Bitmaps. Sie können ein Bitmap von vielen Quellen aus konstruieren wie zum Beispiel Dateien, Bitmap-Handles, Anwendungs-Ressourcen u.a..
- TRegion definiert einen Bereich in einem Fenster. Sie können einen Bereich in vielen verschiedenen Formen konstruieren wie zum Beispiel Rechtecke, Ellipsen und Polygonen (Vielecken). Beachten Sie, daß *TRegion* ein pseudo-GDI-Objekt ist; d.h.: es ist nicht von *TGdiObject* abgeleitet.
- TIcon kapselt Windows-Symbole. Sie können ein Symbol von einer Ressource oder einer expliziten Information aus konstruieren. Beachten Sie, daß *TIcon* ein pseudo-GDI-Objekt ist.
- TCursor kapselt den Windows-Cursor. Sie können den Cursor von einer Ressource oder einer expliziten Information aus konstruieren.
- TDib kapselt die Geräte-unabhängige Bitmap-Klasse (DIB). DIBs besitzen kein Windows-Handle; stattdessen sind sie nur eine Struktur mit Format- und Paletteninformationen und mit einer Kollektion von Bits (Pixeln). Diese Klasse bietet einen bequemen Weg, um mit DIBs wie mit jedem anderen GDI-Objekt zu arbeiten. Ein DIB ist das, was tatsächlich in einer .BMP-Datei, in Bitmap-Ressourcen, steckt und was in die Zwischenablage als ein DIB gelegt wird. Beachten Sie, daß *TDib* ein pseudo-GDI-Objekt ist.

Änderungen an gekapselten GDI-Funktionen

Ihnen werden viele Funktionen in den ObjectWindows-GDI-Klassen vertraut vorkommen. Der Grund dafür liegt darin, daß viele von ihnen dieselben Namen und sehr ähnliche Funktionsköpfe wie reguläre Windows-API-Funktionen besitzen. Da die ObjectWindows-GDI-Klassen die Funktionalität so vieler Windows-Objekte nachbilden, müssen die Klassen nicht mit neuer Terminologie und Fachsprache geladen werden. Deshalb sind die Funktionsnamen und -köpfe mit Bedacht so ähnlich wie mög-

lich, wie Sie das von den Standard-Windows-GDI-Funktionen gewohnt sind.

Es wurden allerdings auch einige Änderungen zur Erleichterung vorgenommen. Diese umfassen das Extrahieren gepackter Rückgabewerte oder die Verwendung von ObjectWindows-Objekten an Stelle von Windows-definierten Strukturen. Viele dieser Änderungen werden unten beschrieben.

Es sollte noch erwähnt werden, daß keine dieser Konvertierungen harte und schnelle Regeln darstellen; nur weil eine Funktion irgendwie konvertiert werden kann, bedeutet das nicht, daß sie es auch notwendigerweise wurde. Wenn Sie aber eine ObjectWindows-Funktion mit demselben Namen wie eine Windows-API-Funktion sehen, die etwas unterschiedlich aussieht, müßte Ihnen einer der folgenden Gründe die Änderung erklären:

- API-Funktionen, die ein Objekt-Handle als ein Parameter erwarten, lassen das Handle in der ObjectWindows-Version oft aus. Das *TGdiObject*-Basisobjekt verwaltet ein Handle auf jedes Objekt. Die ObjectWindows-Version benutzt dann dieses Handle, wenn der Aufruf an Windows übergeben wird. Wenn Sie zum Beispiel ein Objekt in einem Gerätekontext aussuchen, würden Sie normalerweise die API-Funktion *SelectObject* verwenden. Das folgende Beispiel zeigt die Verwendung dieser Funktion:

```
void SelectPen(HDC& hdc, HPEN& hpen)
  HPEN hpenOld;
  hpenOld = SelectObject(hdc, hpen);

  // Führe irgendetwas mit dem neuen Stift durch.
  .
  .
  .

  // Gehe jetzt zum alten Stift zurück.
  SelectObject(hdc, hpenOld);
  }
```

Die ObjectWindows-Version dieser Funktion ist in der Klasse *TDC*, die *TGdiObject* von abgeleitet ist, gekapselt. Hier nun ein Beispiel, wie die oben stehende Funktion in einer Elementfunktion in einer von *TDC* abgeleiteten Klasse auftreten würde. Achten Sie auf den Unterschied zwischen den beiden Aufrufen von *SelectObject*:

```
void SelectPen(TDC& dc, TPEN& pen)
  dc.SelectObject(pen);

  // Führe irgendetwas mit dem neuen Stift durch.
  .
  .
  .

  // Gehe jetzt zum alten Stift zurück.
  dc.RestorePen();
  }
```

- ObjectWindows-GDI-Funktionen setzen üblicherweise einen ObjectWindows-Typ an die Stelle eines Windows-Typs:

  Windows-API-Funktionen verwenden individuelle Parameter, um x- und y-Koordinatenwerte anzugeben; ObjectWindows-GDI-Funktionen verwenden *TPoint*-Objekte.

  Windows-API-Funktionen verwenden RECT-Strukturen, um einen rechteckigen Bereich anzugeben; ObjectWindows-GDI-Funktionen verwenden *TRect*-Objekte.

  Windows-API-Funktionen verwenden RGN-Strukturen, um einen Bereich anzugeben; ObjectWindows-GDI-Funktionen verwenden *TRegion*-Objekte.

  Windows-API-Funktionen erwarten die Parameter HLOCAL oder HGLOBAL, um ein Objekt zu übergeben, das keine vordefinierte Windows-Struktur besitzt; ObjectWindows-GDI-Funktionen verwenden Referenzen auf ObjectWindows-Objekte.

Einige Windows-Funktionen geben ein DWORD mit darin kodierten Daten zurück. Das DWORD muß dann bearbeitet werden, um die Daten daraus zu extrahieren. Die ObjectWindows-Versionen dieser Funktionen erwarten eine Referenz auf ein passendes Objekt als Parameter. Die Funktion plaziert dann die Daten in das Objekt, wodurch der Programmierer von der Aufgabe, den Wert zu extrahieren befreit wird. Diese Funktionen geben normalerweise ein BOOL zurück, der angibt, ob der Funktionsaufruf erfolgreich war.

Die Windows-Version von *SetViewportOrg* zum Beispiel gibt ein DWORD, in dem der alte Wert für den Viewport-Ursprung enthalten ist, zurück. Die ObjectWindows-Version von *SetViewportOrg* erwartet eine *TPoint*-Referenz an Stelle der zwei **int**-Werte, die die Windows-Version als Parameter erwartet. Sie erwartet zudem einen zweiten Parameter, ein *TPoint* *, in dem die alten Viewport-Ursprünge plaziert werden.

Arbeiten mit Gerätekontexten

Wenn Sie mit dem Windows-GDI arbeiten, greifen Sie auf alle Geräte, von Fenstern bis zu Druckern und Plottern, über den Gerätekontext zu. Der Gerätekontext ist eine Struktur, die durch das GDI verwaltet wird. Das GDI enthält wichtige Informationen über das Gerät, mit dem Sie arbeiten, wie zum Beispiel die Standardvordergrund- und Standardhintergrundfarben, die Schriftart, Palette usw.. ObjectWindows 2.5 kapselt Gerätekontextinformation in einer Reihe von DC-Klassen, von denen alle auf der Klasse *TDC* basieren.

TDC enthält die meiste eventuell benötigte Gerätekontext-Funktionalität. Die anderen DC-bezogenen Klassen sind von *TDC* oder *von TDC* abgeleiteten Klassen abgeleitet. Diese abgeleiteten Klassen spezialisieren nur die Funktionalität der Klasse *TDC* und wenden sie auf eine diskrete Menge von Operationen an. Hier ist eine Beschreibung von jeder dieser Gerätekontext-Klassen:

- *TDC* ist die Stammklasse aller GDI-Gerätekontexte für ObjectWindows 2.5; sie kann selbst instanziiert werden oder spezialisierte Subklassen werden für spezielles Verhalten benutzt.
- TWindowDC bietet den Zugriff auf den gesamten, von einem Fenster gehaltenen Bereich; dies ist die Basis für jede DC-Klasse, die ihr Handle nach getaner Arbeit freigibt.
- TScreenDC bietet direkten Zugriff auf die Bildschirm-Bitmap mit Hilfe eines Gerätekontextes für das Fenster-Handle 0, welches für den ganzen Bildschirm ohne Ausschnitte steht.
- TDesktopDC bietet Zugriff auf den Client-Bereich des Desktop-Fensters, welches der Bildschirm hinter allen anderen Fenstern ist.
- TClientDC bietet Zugriff auf den zu einem Fenster gehörigen Client-Bereich.
- TPaintDC hüllt Aufrufe von *BeginPaint* und *EndPaint* zur Verwendung in einer WM_PAINT-Antwortfunktion ein.
- TMetaFileDC bietet einen Gerätekontext mit einer zur Verwendung geladenen Metadatei.
- TCreatedDC ermöglicht das Erzeugen eines Gerätekontextes für ein angegebenes Gerät.
- TIC ermöglicht das Erzeugen eines Informationskontextes für ein angegebenes Gerät.
- TMemoryDC bietet Zugriff auf einen Speiter-Gerätekontext.
- TDibDC bietet Zugriff auf DIBs unter Verwendung der Treibers DIB.DRV.
- TPrintDC bietet Zugriff auf einen Drucker-Gerätekontext.

Die Klasse TDC

Während die spezialisierten Gerätekontextklassen eine extra Funktionalität bereitstellen, die für die speziellen Absichten jeder Klasse zugeschneidert sind, bietet die Klasse *TDC* das *meiste* der Funktionalität von jeder Klasse. Dieser Abschnitt beschreibt diese Basisfunktionalität.

Wegen der großen Zahl der in *TDC* enthaltenen Funktionen wird nicht jede Funktion im Detail hier beschrieben; stattdessen werden Funktionalitätsbereiche, die in der Klasse *TDC* enthalten sind, beschrieben und zusätzlich nur einige Funktionen speziell erläutert. Eine Erklärung zu den anderen findet sich im ObjectWindows *Referenzhandbuch*. Insbesondere besitzen viele *TDC*-Funktionen ein ähnliches Aussehen wie entsprechende Windows-API-Funktionen. Sie werden deshalb nicht eingehender be-

trachtet. ObjectWindows-specifische Funktionen und die wichtigsten API-ähnlichen Funktionen sind in diesem Abschnitt beschrieben.

Konstruieren und Entfernen von TDC

TDC bietet einen öffentlichen Konstruktor und einen öffentlichen Destruktor. Der öffentliche Konstruktor erwartet ein HDC, ein Handle auf einen Gerätekontext. Das bedeutet letztlich, daß Sie einen existierenden Gerätekontext bereits haben müssen, bevor Sie ein *TDC*-Objekt konstruieren können. Normalerweise konstruieren Sie ein *TDC* nicht direkt, auch wenn Sie es könnten. Stattdessen verwenden Sie im allgemeinen das *TDC*-Objekt, wenn Sie einen Gerätekontext als einen Funktionsparameter oder einen Zeiger auf ein *TDC* übergeben, um auf einen Gerätekontext zu zeigen, der in *TDC* oder in einem von *TDC* abgeleiteten Objekt enthalten ist.

~TDC stellt alle Standardobjekte im Gerätekontext wieder her und wirft die Objekte weg.

TDC bietet ebenso zwei geschützte Konstruktoren für eine Verwendung durch abgeleitete Klassen. Der erste ist ein Standardkonstruktor, der verhindert, daß abgeleitete Klassen den Konstruktor von *TDC* explizit aufrufen. Der zweite Konstruktor erwartet ein HDC und ein *TAutoDelete*-Flag. *TAutoDelete* ist ein **enum**-Wert, der *NoAutoDelete* oder *AutoDelete* annehmen kann. Der Parameter *TAutoDelete* wird für die Initialisierung des Elementes *ShouldDelete* verwendet, das von *TGdiObject* geerbt ist (der öffentliche Konstruktor von *TDC* initialisiert es mit *NoAutoDelete*).

DC-Operatoren

TDC bietet einen Konvertierungsoperator HDC. HDC ermöglicht Ihnen die Rückgabe des Handle auf den Gerätekontext Ihres speziellen *TDC*- bzw. von *TDC* abgeleiteten Objektes. Dieser Operator wird meistens implizit ausgeführt. Wenn Sie ein *TDC*-Objekt verwenden, wo Sie normalerweise ein HDC benutzen würden (zum Beispiel in einem Funktionsaufruf o.ä.), versucht der Compiler einen Weg zu finden, das Objekt auf den erforderlichen Typ umzuwandeln. Damit verwendet er den HDC-Konvertierungsoperator, auch wenn er nicht explizit aufgerufen ist.

Nehmen wir zum Beispiel an, Sie wollen einen Gerätekontext im Speicher erzeugen, der mit dem mit einem *TDC*-Objekt assoziierten Gerät kompatibel ist. Sie können die Windows-API-Funktion *CreateCompatibleDC* verwenden, um den neuen DC von Ihrem existierenden *TDC*-Objekt aus zu erzeugen.

```
HDC GetCompatDC(TDC& dc, TWindow& window)
{
  HDC compatDC;
```

```
  if(!(compatDC = CreateCompatibleDC(dc)))
  {
    window.MessageBox("Couldn't create compatible DC!", "Failure",
                      MB_OK | MB_ICONEXCLAMATION);
    return NULL;
  }
  else return compatDC;
}
```

Beachten Sie, daß *CreateCompatibleDC* einen einzigen Parameter, ein HDC, erwartet. Damit wird der Funktionsparameter *dc* im Aufruf *CreateCompatibleDC* von implizit zu HDC umgewandelt.

DC-Funktionen

Die Funktionen in diesem Abschnitt dienen dem Zugriff auf Informationen über den Gerätekontext selbst. Sie sind äquivalent zu den gleichnamigen Windows-API-Funktionen.

Sie können über die Funktionen *SaveDC* und *RestoreDC* einen Gerätekontext ganz normal speichern und wiederherstellen. Das folgende Code-Beispiel zeigt eine mögliche Verwendung dieser Funktionen. Beachten Sie, daß für den einzigen Parameter von *RestoreDC* kein **int**-Wert sondern ein Standardwert angegeben wird:

```
void
TMyDC::SomeFunc(TDC& dc, int x1, int y1, int x2, int y2)
{
  dc.SaveDC();

  dc.SetMapMode(MM_LOENGLISH);
  :
  dc.Rectangle(x1, -y1, x2, -y2);
  dc.RestoreDC();
}
```

Sie können ferner einen Gerätekontext auf die Einstellungen in einer Struktur DEVMODE mit Hilfe der Funktion *ResetDC* zurücksetzen. Der einzige Parameter von *ResetDC* erwartet eine Referenz auf eine DEVMODE-Struktur.

Über die Funktion *GetDeviceCaps* können Sie gerätespezifische Information über ein gegebenes Anzeigegerät erhalten. Diese Funktion erwartet einen Parameter, ein **int**-Index auf den Typ von Information, die vom Gerätekontext geliefert werden soll. Die möglichen Werte für diesen Parameter sind dieselben wie bei der Windows-API-Funktion.

Mit Hilfe der Funktion *GetDCOrg* können Sie die aktuellen logischen Koordinaten des Gerätekontextes innerhalb der absoluten physikalischen Koordinaten des Anzeigegerätes festlegen.

Diese Funktion erwartet eine Referenz auf eine *TPoint*-Struktur und gibt ein BOOL zurück. Der BOOL-Wert zeigt an, ob der Funktionsaufruf erfolgreich war. Dabei enthält das *TPoint*-Objekt die Koordinaten für den versetzen Ursprung des Gerätekontextes.

Auswählen und Wiederherstellen von GDI-Objekten

Über die Funktion *SelectObject* können Sie ein GDI-Objekt in einen Gerätekontext plazieren. Es existieren vier Versionen der Funktion *SelectObject*, die alle **void** zurückgeben, aber unterschiedliche Parameter erwarten. Welche Funktion Sie verwenden, hängt vom Objekttyp ab, der für den Gerätekontext ausgewählt werden soll. Die verschiedenen Versionen sind:

```
SelectObject(const TBrush& brush);
SelectObject(const TPen& pen);
SelectObject(const TFont& font);
SelectObject(const TPalette& palette, BOOL forceBG=FALSE);
```

Zusätzlich ermöglicht Ihnen *TMemoryDC* die Auswahl einer Bitmap.

Grafikobjekte, die Sie für einen Gerätekontext auswählen können, existieren normalerweise als logische Objekte, die die zur Erzeugung des Objektes erforderliche Information enthalten. Die Grafikobjekte sind über ein Windows-Handle mit den logischen Objekten verbunden. Wenn ein Grafikobjekt für den Gerätekontext ausgewählt ist, wird ein physikalisches Werkzeug mit Hilfe der im logischen Stift enthaltenen Attribute innerhalb des Gerätekontextes erzeugt.

Über die Funktion *SelectStockObject* können Sie auch ein vordefiniertes Objekt auswählen. *SelectStockObject* erwartet einen Parameter, ein **int**, der äquivalent zum Parameter für den Aufruf der API-Funktion *GetStockObject* ist. Im wesentlichen finden mit der Funktion *SelectStockObject* zwei Aufrufe statt: ein Aufruf von *GetStockObject*, um ein vordefiniertes Objekt anzufordern, und dann ein Aufruf von *SelectObject*, um das vordefinierte Objekt im Gerätekontext zu plazieren.

TDC bietet Funktionen für das Wiederherstellen originaler Objekte in einem Gerätekontext. Dazu existieren im wesentlichen vier Versionen: *RestoreBrush*, *RestorePen*, *RestoreFont* und *RestorePalette*. Eine fünfte, *RestoreTextBrush*, existiert nur für 32-Bit-Anwendungen. Zudem existiert noch eine andere Funktion, *RestoreObjects*, die alle vier Funktionen (oder fünf bei 32-Bit) aufruft, um alle originalen Objekte im Gerätekontext wiederherzustellen. Alle diese Funktionen geben **void** zurück und erwarten keine Parameter.

Hilfsfunktionen zum Zeichnen

GetBrushOrg erwartet als einzigen Parameter eine Referenz auf ein *TPoint*-Objekt. Es plaziert die Koordinaten des Pinselursprungs in das

TPoint-Objekt. *GetBrushOrg* gibt TRUE zurück, wenn die Operation erfolgreich war.

SetBrushOrg erwartet zwei Parameter, eine Referenz auf ein *TPoint*-Objekt und ein *TPoint**. Die Funktion setzt den Pinselursprung des Gerätekontextes auf die *x*- und *y*-Werte aus dem ersten *TPoint*-Objekt. Wenn Sie beim zweiten Parameter keinen Wert angeben, wird standardmäßig Null angenommen. Wenn Sie einen Zeiger auf ein *TPoint*-Objekt als zweiten Parameter übergeben, legt *TDC::SetBrushOrg* die alten Werte für den Pinselursprung in die Elemente *x* und *y* des Objektes ab. Der Rückgabewert zeigt an, ob die Operation erfolgreich war.

Farb- und Palettenfunktionen

TDC enthält eine Reihe von Funktionen, mit denen Sie die Farben und die Palette eines Gerätekontextes manipulieren können.

```
GetNearestColor                GetSystemPaletteEntries
RealizePalette                 GetSystemPaletteUse
UpdateColors                   SetSystemPaletteUse
```

Funktionen für Attribute zum Zeichnen

Verwenden Sie Funktionen für Attribute zum Zeichnen, um den Zeichenmodus des Gerätekontextes zu setzen. Alle diese Funktionen sind zu den gleichnamigen API-Funktionen analog, mit der Ausnahme, daß der HDC-Parameter in jeder Funktion fehlt.

```
GetBkColor                     GetROP2
SetBkColor                     SetROP2
GetBkMode                      GetStretchBltMode
SetBkMode                      SetStretchBltMode
GetPolyFillMode                GetTextColor
SetPolyFillMode                SetTextColor
```

Eine weitere Funktion, *SetMiterLimit*, ist nur für 32-Bit-Anwendungen verfügbar.

Viewport- und Fenster-Abbildungsfunktionen

Verwenden Sie die folgenden Funktionen, um die Viewport- und Fenster-Abbildungsmodi zu setzen:

```
GetMapMode                     ScaleViewportExt
SetMapMode                     GetWindowOrg
GetViewportOrg                 GetWindowOrg
GetViewportOrg                 OffsetWindowOrg
SetViewportOrg                 SetWindowOrg
OffsetViewportOrg              GetWindowExt
GetViewportExt                 GetWindowExt
GetViewportExt                 SetWindowExt
SetViewportExt                 ScaleWindowExt
```

Die folgenden Viewport- und Fenster-Abbildungsfunktionen sind nur bei 32-Bit-Anwendungen verfügbar:

```
ModifyWorldTransform                SetWorldTransform
```

Koordinatenfunktionen

Koordinatenfunktionen konvertieren logische Koordinaten in physikalische Koordinaten und umgekehrt:

```
DPtoLP                              LPtoDP
```

Funktionen zum Clipping, für Bereiche und zum Aktualisieren von Rechtecken

Verwenden Sie Funktionen zum Clipping, für Bereiche und zum Aktualisieren von Rechtecken, um einfache oder komplexe Bereiche im Clipping-Bereich eines Gerätekontextes einzurichten und wiederherzustellen:

```
GetClipBox                          PtVisible
ExcludeClipRect                     RectVisible
ExcludeUpdateRgn                    GetBoundsRect
IntersectClipRect                   SetBoundsRect
OffsetClipRgn                       GetClipRgn
SelectClipRgn
```

MetaFile-Funktionen

Mit MetaFile-Funktionen können Sie auf Metadateien zugreifen:

```
EnumMetaFile                        PlayMetaFileRecord
PlayMetaFile
```

Funktionen zur aktuellen Position

Verwenden sie diese Funktionen, um den aktuellen Punkt im Gerätekontext zu bewegen. Es existieren drei Versionen von *MoveTo*:

- *MoveTo*(**int** *x*, **int** *y*) bewegt den Stift zum Punkt *x*, *y*.
- *MoveTo*(*TPoint &point*) bewegt den Stift zum Punkt *point.x*, *point.y*.
- *MoveTo*(*TPoint &point*, *TPoint &oldPoint*) bewegt den Stift zum Punkt *point.x*, *point.y* und legt die alte Stiftposition in *oldPoint* ab.

GetCurrentPosition erwartet eine Referenz auf ein *TPoint*-Objekt. Sie legt die Koordinaten der aktuellen Position im *TPoint*-Objekt ab und gibt TRUE zurück, wenn der Funktionsaufruf erfolgreich war.

Schriftfunktionen

Verwenden Sie die Schriftfunktionen von *TDC*, um auf Schriften zuzugreifen und sie zu manipulieren:

```
GetCharWidth                        EnumFontFamilies
SetMapperFlags                      GetFontData
GetAspectRatioFilter                GetCharABCWidths
EnumFonts
```

Pfadfunktionen

Pfadfunktionen sind nur in 32-Bit-Anwendungen verfügbar. Die Pfadfunktionen von *TDC* sind dieselben wie die Versionen zu Win32, mit der Ausnahmen, daß die Versionen von *TDC* keinen HDC-Parameter erwarten.

```
BeginPath                    FillPath
CloseFigure                  StrokePath
EndPath                      StrokeAndFillPath
FlattenPath                  SelectClipPath
WidenPath                    PathToRegion
```

Ausgabefunktionen

TDC bietet eine Vielzahl von Ausgabefunktionen für alle verschiedenen Arten von Objekten, die ein Standardgerätekontext behandeln kann, einschließlich:

- Symbolen
- Rechtecken
- Bereichen
- Formen
- Bitmaps
- Text

Fast alle diese Funktionen besitzen mehrere Versionen: eine Version bietet eine nahezu identische Funktionalität zu der korrespondierenden API-Funktion, mit der Ausnahme, daß der Parameter HDC fehlt. Andere Versionen verwenden *TPoint*, *TRect*, *TRegion* und andere ObjectWindows Datenkapselungen, die Aufrufe kurz und verständlicher machen. Diese Funktionen werden im ObjectWindows *Referenzhandbuch* eingehender beschrieben.

- **Aktuelle Position**

```
GetCurrentPosition           MoveTo
```

- **Symbole**

```
DrawIcon
```

- **Rechtecke**

```
FrameRect                    DrawFocusRect
FillRect                     TextRect
InvertRect
```

- **Bereiche**

```
FrameRgn                     InvertRgn
FillRgn                      PaintRgn
```

Die folgenden Funktionen sind nur bei 32-Bit-Anwendungen verfügbar:

- **Formen**

```
Arc                          Polyline
Chord                        Polygon
```

```
Pie                                PolyPolygon
Ellipse                            Rectangle
LineTo                             RoundRect
LineDDA
```

- **Bitmaps und Blitting**

```
GetPixel                           GetDIBits
SetPixel                           SetDIBits
BitBlt                             SetDIBitsToDevice
PatBlt                             FloodFill
ScrollDC                           ExtFloodFill
StretchBlt                         StretchDIBits
```

- **Text**

```
TextOut                            DrawText
ExtTextOut                         GrayString
TabbedTextOut
```

Die folgenden Funktionen sind nur für 32-Bit-Anwendungen verfügbar:

- **Formen**

```
AngleArc                           PolyBezierTo
PolyPolyline                       PolyDraw
PolyBezier                         PolylineTo
```

- **Bitmaps und Blitting**

```
MaskBlt                            PlgBlt
```

Objektdatenelemente und Funktionen

Diese Datenelemente und Funktionen werden benutzt, um das Gerätekontextobjekt selbst zu verwalten. Alle in diesem Abschnitt beschriebenen Funktionen und Datenelemente sind geschützt und können nur durch eine von *TDC* abgeleitete Klasse zugegriffen werden.

- *ShouldDelete* gibt an, ob das Objekt sein Handle auf den Gerätekontext löschen sollte, wenn der Destruktor aufgerufen wird.
- Handle enthält das aktuelle Handle auf den Gerätekontext.
- OrgBrush, *OrgPen*, *OrgFont* und *OrgPalette* sind Handles auf die originalen Objekte, wenn der Gerätekontext erzeugt wurde; *OrgTextBrush* ist auch in 32-Bit-Anwendungen vorhanden
- CheckValid löst eine Exception aus, wenn das Gerätekontextobjekt nicht gültig ist.
- Init setzt die Werte von *OrgBrush*, *OrgPen*, *OrgFont* und *OrgPalette* beim Erzeugen des Objektes; wenn Sie eine von *TDC* abgeleitete Klasse erzeugen, ohne den Konstruktor von *TDC* explizit aufzurufen, sollten Sie *TDC::Init* als erstes in Ihrem Konstruktor aufrufen.
- GetHDC gibt über *Handle* ein HDC zurück.
- GetAttributeHDC gibt wie *GetHDC* über *Handle* ein HDC zurück; wenn Sie ein Objekt mit mehr als einem Gerätekontext erzeugen, sollten Sie

diese Funktion und nicht *GetHDC* überschreiben, um eine geeignete Rückgabe zu erhalten.

- OWLFastWindowFrame zeichnet einen Rahmen, der oft als Fensterrahmen verwendet wird. Diese Funktion greift auf die nicht-dokumentierte Windows-API-Funktion *FastWindowFrame* zurück, falls verfügbar, bzw. auf *PatBlt*, falls nicht verfügbar.

Die Klasse TPen

Die Klasse *TPen* kapselt den logischen Stift. Sie enthält eine Farbe für die »Tinte« des Stiftes (in einem *TColor*-Objekt gekapselt), eine Stiftbreite und den Stiftstil.

Konstruieren von TPen

Sie können ein *TPen* entweder direkt mit Angabe der Farbe, der Breite und dem Stil des Stiftes oder indirekt mit Angabe eines *TPen* **&** oder eines Zeigers auf eine LOGPEN-Struktur konstruieren. Das direkte Konstruieren eines Stiftes erzeugt ein neues Objekt mit den angegebenen Attributen. Hier ist der Konstruktor für das direkte Konstruieren eines Stiftes:

```
TPen(TColor color, int width=1, int style=PS_SOLID);
```

Der Parameter *style* kann einen der folgenden Werte annehmen: PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT, PS_NULL oder PS_INSIDERFRAME. Diese Werte werden im ObjectWindows *Referenzhandbuch* beschrieben.

Das indirekte Konstruieren eines Stiftes erzeugt ein neues Objekt, kopiert aber die Attribute des übergebenen Objektes in das neue Stiftobjekt. Hier sind die Konstruktoren für das indirekte Konstruieren eines Stiftes:

```
TPen(const LOGPEN far* logPen);
TPen(const TPen&);
```

Sie können auch ein neues *TPen*-Objekt von einem existierenden HPEN-Handle aus erzeugen:

```
TPen(HPEN handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, um ein ObjectWindows-Objekt als Alias zu einem regulären Windows-Handle, das in einer Botschaft empfangen wurde, zu erhalten.

Es gibt auch noch zwei andere Konstruktoren, die nur für 32-Bit-Anwendungen verfügbar sind. Sie können diese Konstruktoren verwenden, um kosmetische und geometrische Stifte zu erzeugen:

```
TPen(DWORD penStyle,
     DWORD width,
     const TBrush& brush,
     DWORD styleCount,
     LPDWORD style);
TPen(DWORD penStyle,
     DWORD width,
     const LOGBRUSH& logBrush,
     DWORD styleCount,
     LPDWORD style);
```

Dabei gilt:

- *penStyle* ist eine Kombination aus Typ, Stil, End Cap und Join des Stiftes ist, wobei gilt:
  - Typ ist entweder PS_GEOMETRIC oder PS_COSMETIC.
  - Stil kann einen der folgenden Werte annehmen:

```
PS_ALTERNATE                    PS_SOLID
PS_DASH                         PS_DOT
PS_DASHDOT                      PS_DASHDOTDOT
PS_NULL                         PS_USERSTYLE
PS_INSIDEFRAME
```

  - End Cap wird nur bei geometrischen Stiften angegeben und kann einen der folgenden Werte annehmen:

```
PS_ENDCAP_ROUND                 PS_ENDCAP_SQUARE
PS_ENDCAP_FLAT
```

  - Join wird nur bei geometrischen Stiften angegeben und kann einen der folgenden Werte annehmen:

```
PS_JOIN_BEVEL                   PS_JOIN_MITER
PS_JOIN_ROUND
```

- *width* ist die Stiftbreite.
- brush oder *logBrush* ist eine Referenz auf ein existierendes *TBrush*- oder LOGBRUSH-Objekt.
- styleCount ist die Größe (in DWORDs) des *style*-Array; *styleCount* sollte Null sein, es sei denn der Stiftstil ist PS_USERSTYLE.
- style ist ein Zeiger auf ein Array mit DWORDs, das das Muster des Stiftes festlegt; *style* sollte NULL sein, es sei denn, der Stiftstil ist PS_USERSTYLE

Zugriffe auf TPen

Sie können auf *TPen* über eine HPEN als eine LOGPEN-Struktur zugreifen. Um ein HPEN von einem *TPen*-Objekt zu erhalten, verwenden Sie den HPEN-Operator mit dem *TPen*-Objekt als Parameter. Der HPEN-Operator wird fast nie explizit aufgerufen:

```
HPEN GetHPen(TPen& pen)
{
  return pen;
}
```

Dieser Code ruft automatisch den Konvertierungsoperator HPEN auf, um das *TPen*-Objekt in den korrekten Typ umzuwandeln.

Um ein *TPen*-Objekt in eine LOGPEN-Struktur umzuwandeln, verwenden Sie die Funktion *GetObject*:

```
BOOL
GetLogPen(LOGPEN far& logPen)
{
  TPen pen(TColor::LtMagenta, 10);
  return pen.GetObject(logPen);
}
```

Das folgende Beispiel zeigt die Verwendung eines Stiftes mit einem *TDC* zum Ziehen eines Linie:

```
void
TPenDemo::DrawLine(TDC& dc, const TPoint& point, TColor& color)
{
  TPen BrushPen(color, PenSize);
  dc.SelectObject(BrushPen);
  dc.LineTo(point);
}
```

Die Klasse TBrush

Die Klasse *TBrush* kapselt den logischen Pinsel. Sie enthält eine Farbe für die Tinte des Pinsels (gekapselt in einem *TColor*-Objekt), eine Pinselbreite und abhängig von der Art, wie der Pinsel konstruiert wurde, einen Pinselstil, -muster oder -Bitmap.

Konstruieren von TBrush

Sie können ein *TBrush* entweder direkt mit Angabe der Farbe, der Breite und dem Stil des Pinsels oder indirekt mit Angabe eines *TBrush* **&** oder eines Zeigers auf eine LOBRUSH-Struktur konstruieren. Das direkte Konstruieren eines Pinsels erzeugt ein neues Objekt mit den angegebenen Attributen. Hier sind die Konstruktoren für das direkte Konstruieren eines Pinsels:

```
TBrush(TColor color);
TBrush(TColor color, int style);
TBrush(const TBitmap& pattern);
TBrush(const TDib& pattern);
```

Der erste Konstruktor erzeugt einen soliden Pinsel mit der in *color* enthaltenen Farbe.

Der zweite Konstruktor erzeugt einen Schraffurpinsel mit der in *color* enthaltenen Farbe und einem in *style* enthaltenen Schraffurstil. *style* kann eine der folgenden Werte annehmen:

```
HS_BDIAGONAL                    HS_CROSS
HS_DIAGCROSS                    HS_FDIAGONAL
HS_HORIZONTAL                   HS_VERTICAL
```

Der dritte und vierte Konstruktor erzeugt einen mit einer als Parameter übergebenen Bitmap oder DIB arbeitenden Pinsel. Die Pinselbreite hängt von der Breite der Bitmap oder der DIB ab.

Das indirekte Konstruieren eines Pinsels erzeugt ein neues Objekt, kopiert aber die Attribute des übergebenen Objektes in das neue Pinselobjekt. Hier sind die Konstruktoren für das indirekte Konstruieren eines Pinsels:

```
TBrush(const LOGBRUSH far* logBrush);
TBrush(const TBrush& src);
```

Sie können ein neues *TBrush*-Objekt auch von einem existierenden HBRUSH-Handle aus erzeugen:

```
TBrush(HBRUSH handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, um ein ObjectWindows-Objekt als Alias zu einem regulären Windows-Handle, das in einer Botschaft empfangen wurde, zu erhalten.

Zugriffe auf *TBrush*

Sie können auf *TBrush* über eine HBRUSH als eine LOGBRUSH-Struktur zugreifen. Um ein HBRUSH von einem *TBrush*-Objekt zu erhalten, verwenden Sie den HBRUSH-Operator mit dem *TBrush*-Objekt als Parameter. Der HBRUSH-Operator wird fast nie explizit aufgerufen:

```
HBRUSH GetHBrush(TBrush& brush)
{
  return brush;
}
```

Dieser Code ruft automatisch den Konvertierungsoperator HBRUSH auf, um das *T*Brush-Objekt in den korrekten Typ umzuwandeln.

Um ein *TBrush*-Objekt in eine LOGBRUSH-Struktur umzuwandeln, verwenden Sie die Funktion *GetObject*:

```
BOOL GetLogBrush(LOGBRUSH far& logBrush)
{
  TBrush brush(TColor::LtCyan, HS_DIAGCROSS);
  return brush.GetObject(logBrush);
}
```

Um den Ursprung eines Pinselobjektes zurückzusetzen, verwenden Sie die Funktion *UnrealizeObject*. *UnrealizeObject* setzt den Pinselursprung zurück und gibt bei Erfolg einen Wert ungleich Null zurück.

Das folgende Beispiel zeigt die Verwendung eines Pinsels, um ein Rechteck in einem Fenster zu malen:

```
void
TMyWindow::PaintRect(TDC& dc, TPoint& p, TSize& size)
{
  TBrush brush(TColor(5,5,5));
  dc.SelectObject(brush);
  dc.Rectangle(p, size);
  dc.RestoreBrush();
}
```

Die Klasse TFont

Die Klasse *TFont* ermöglicht ein einfaches Erzeugen und Verwenden von Windows-Schriften in Ihren Anwendungen. Die Klasse *TFont* kapselt alle Attribute für eine logische Schrift.

Konstruieren von TFont

Sie können ein *TFont* entweder direkt mit Angabe der Schriftattribute im Konstruktor oder indirekt mit Angabe eines *TFont* & oder eines Zeigers auf eine LOFONT-Struktur konstruieren. Das direkte Konstruieren einer Schrift erzeugt ein neues Objekt mit den angegebenen Attributen. Hier sind die Konstruktoren für das direkte Konstruieren einer Schrift:

```
TFont(const char far* facename=0,
      int height=0, int width=0, int escapement=0,
      int orientation=0, int weight=FW_NORMAL,
      BYTE pitchAndFamily=DEFAULT_PITCH|FF_DONTCARE,
      BYTE italic=FALSE, BYTE underline=FALSE,
      BYTE strikeout=FALSE,
      BYTE charSet=1,
      BYTE outputPrecision=OUT_DEFAULT_PRECIS,
      BYTE clipPrecision=CLIP_DEFAULT_PRECIS,
      BYTE quality=DEFAULT_QUALITY);

TFont(int height, int width, int escapement=0,
      int orientation=0,
      int weight=FW_NORMAL,
      BYTE italic=FALSE, BYTE underline=FALSE,
      BYTE strikeout=FALSE,
      BYTE charSet=1,
      BYTE outputPrecision=OUT_DEFAULT_PRECIS,
      BYTE clipPrecision=CLIP_DEFAULT_PRECIS,
```

```
    BYTE quality=DEFAULT_QUALITY,
    BYTE pitchAndFamily=DEFAULT_PITCH|FF_DONTCARE,
    const char far* facename=0);
```

Der erste Konstruktor dient dazu, die am häufigsten verwendeten Attribute für eine Schrift wie zum Beispiel den Namen, die Höhe, die Breite usw. auf bequeme Weise einzubauen und bei den anderen Attributen, die im allgemeinen immer dieselben Werte besitzen, die Vorgabewerte zu übernehmen. Der zweite Konstruktor hat seine Parameter in derselben Reihenfolge wie der Windows-API-Aufruf *CreateFont*, damit Sie Fragmente eines bereits existierenden Windows-Codes auf einfache Weise übernehmen können.

Das indirekte Konstruieren einer Schrift erzeugt ein neues Objekt, kopiert aber die Attribute des übergebenen Objektes in das neue Schriftobjekt. Hier sind die Konstruktoren für das indirekte Konstruieren einer Schrift:

```
TFont(const LOGFONT far* logFont);
TFont(const TFont& src);
```

Sie können ein neues *TFont*-Objekt auch von einem existierenden HFONT-Handle aus erzeugen:

```
TFont(HFONT handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, um ein ObjectWindows-Objekt als Alias zu einem regulären Windows-Handle, das in einer Botschaft empfangen wurde, zu erhalten.

Zugriffe auf TFont

Sie können auf *TFont* über eine HFONT als eine LOGFONT-Struktur zugreifen. Um ein HFONT von einem *TFont*-Objekt zu erhalten, verwenden Sie den HFONT-Operator mit dem *TFont*-Objekt als Parameter. Der HFONT-Operator wird fast nie explizit aufgerufen:

```
HFONT GetHFont(TFont& font)
{
  return font;
}
```

Dieser Code ruft automatisch den Konvertierungsoperator HFONT auf, um das *TFont*-Objekt in den korrekten Typ umzuwandeln.

Um ein *TFont*-Objekt in eine LOGFONT-Struktur umzuwandeln, verwenden Sie die Funktion *GetObject*:

```
BOOL GetLogFont(LOGFONT far& logFont)
{
  TFont font("Times Roman", 20, 8);
  return font.GetObject(logFont);
}
```

Die Klasse TPalette

Die Klasse *TPalette* kapselt die Windows-Farbpallete, die in Verbindung mit Bitmaps und DIBs verwendet werden kann. *TPalette* ermöglicht Ihnen das Anpassen der Farbtabelle oder individueller Farben, das Bewegen einer Palette in die Zwischenablage und anderes mehr.

Konstruieren von TPalette

Sie können ein *TPalette*-Objekt entweder direkt mit der Übergabe eines Array mit Farben an den Konstruktor oder indirekt mit der Angabe einer *TPalette* **&**, eines Zeigers auf eine LOGPALETTE-Struktur, eines Zeigers auf einen Bitmap-Header usw. konstruieren.

Das direkte Konstruieren einer Palette erzeugt ein neues Objekt mit den angegebenen Attributen. Hier sind die Konstruktoren für das direkte Konstruieren einer Palette:

```
TPalette(const PALETTEENTRY far* entries, int count);
```

entries ist ein Array mit PALETTEENTRY-Objekten. Jedes PALETTEENTRY-Objekt enthält einen Farbwert, der durch drei separate Werte (jeweils einen für rot grün und blau) angegeben ist, sowie eine Variable mit Flags für den Eintrag. *count* gibt die Zahl von Werten im *entries*-Array an.

Das indirekte Konstruieren einer Palette erzeugt ein neues Objekt, kopiert aber die Attribute des übergebenen Objektes in das neue Palettenobjekt. Hier sind die Konstruktoren für das indirekte Erzeugen einer Palette:

```
TPalette(const TClipboard&);
TPalette(const TPalette& palette);
TPalette(const LOGPALETTE far* logPalette);
TPalette(const BITMAPINFO far* info, UINT flags=0);
TPalette(const BITMAPCOREINFO far* core, UINT flags=0);
TPalette(const TDib& dib, UINT flags=0);
```

Jeder dieser Konstruktoren kopiert die Farbwerte aus dem an den Konstruktor übergebenen Objekt in das neue Objekt. Die an den Konstruktor übergebenen Objekte sind nicht notwendig selbst Paletten. Viele von ihnen sind Objekte, die Paletten benutzen und selbst eine Palette enthalten. In diesen Fällen extrahiert der Konstruktor *TPalette* die Palette aus dem Objekt und kopiert sie in das neue Objekt.

Sie können ein neues *TPalette*-Objekt auch von einem existierenden HPALETTE-Handle aus erzeugen:

```
TPalette(HPALETTE handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, um ein ObjectWindows-Objekt als Alias zu einem regulären Windows-Handle, das in einer Botschaft empfangen wurde, zu erhalten.

Zugriffe auf TPalette

Sie können auf *TPalette* über eine HPALETTE oder als eine LOGPALETTE-Struktur zugreifen. Um eine HPALETTE von einem *TPalette*-Objekt zu erhalten, verwenden Sie den HPALETTE-Operator mit dem *TPalette*-Objekt als Parameter. Der HPALETTE-Operator wird fast nie explizit aufgerufen:

```
HPALETTE GetHPalette(TPalette& palette)
{
  return palette;
}
```

Dieser Code ruft den Konvertierungsoperator HPALETTE automatisch auf, um das *TPalette*-Objekt in den korrekten Typ umzuwandeln.

Die Funktion *GetObject* für *TPalette* arbeitet in gleicher Weise wie der Windows-API-Aufruf *GetObject*, wenn ein Handle auf eine Palette übergeben wird: sie legt die Zahl von Einträgen der Farbtabelle in die WORD-Referenz, die an sie als Parameter übergeben wurde, ab. *TPalette::GetObject* gibt bei Erfolg TRUE zurück.

Elementfunktionen

TPalette kapselt ferner eine Reihe von Standard-API-Aufrufen für das Manipulieren von Paletten:

- Sie können eine Farbe mit einem Eintrag in einer Palette über die Funktion *GetNearestPaletteIndex* abgleichen. *GetNearestPaletteIndex* erwartet einen einzigen Parameter, ein *TColor*-Objekt. *GetNearestPaletteIndex* gibt den Index des Eintrags mit der größten Übereinstimmung in der Farbtabelle der Palette zurück.
- GetNumEntries erwartet keine Parameter und gibt die Zahl von Einträgen in der Farbtabelle der Palette zurück.
- Über die Funktion *GetPaletteEntries* erhalten Sie die Werte für einen Bereich von Einträgen in der Farbtabelle der Palette. *TPalette::GetPaletteEntries* arbeitet genauso wie der Windows-API-Aufruf *GetPaletteEntries* mit der Ausnahme, daß der HPALETTE-Parameter bei *TPalette::GetPaletteEntries* entfällt.
- Über die Funktion *SetPaletteEntries* können Sie die Werte für einen Bereich von Einträgen in der Farbtabelle der Palette setzen. *TPalette::SetPaletteEntries* arbeitet genauso wie der Windows-API-Aufruf *SetPaletteEntries* mit der Ausnahme, daß der HPALETTE-Parameter bei *TPalette::SetPaletteEntries* entfällt.
- TPalette bietet ferner zwei Funktionen mit Namen *GetPaletteEntry* und *SetPaletteEntry*. Diese beiden Funktionen arbeiten genauso wie *GetPaletteEntries* und *SetPaletteEntries* mit der Ausnahme, daß sie nur auf einem einzelnen Paletteneintrag arbeiten. Beide Funktionen erwarten zwei Parameter, den Index des Paletteneintrages und eine Referenz auf ein PALETTEENTRY-Objekt. *GetPaletteEntry* legt den Farbwert des gewünsch-

ten Paletteneintrags im PALETTEENTRY-Objekt ab. *SetPaletteEntry* setzt den durch den Index angegebenen Paletteneintrag auf den Wert des PALETTEENTRY-Objektes.

- Über die Funktion *ResizePalette* können Sie die Größe der Palette ändern. *ResizePalette* erwartet einen UINT-Parameter, der die Zahl der Einträge in der neuen Palette festlegt. *ResizePalette* arbeitet genauso wie der Windows-API-Aufruf *ResizePalette*.
- Die Funktion *AnimatePalette* ermöglicht Ihnen das Ersetzen von Einträgen in der Farbtabelle der Palette. *AnimatePalette* erwartet drei Parameter, zwei UINTs und ein Zeiger auf ein Array mit PALETTEENTRY-Objekten. Der erste UINT-Wert gibt den ersten zu ersetzenden Eintrag in der Palette an. Der zweite UINT-Wert legt die Zahl der zu ersetzenden Einträge fest. Die durch diese beiden UINT-Werte bestimmten Einträge werden durch die Werte aus dem Array mit PALETTEENTRYs ersetzt.
- Sie können ferner die Funktion *UnrealizeObject* für Ihr Palettenobjekt verwenden. *UnrealizeObject* paßt die Palette an die aktuelle Systempalette an. *UnrealizeObject* erwartet keine Parameter und arbeitet genauso wie der Windows-API-Aufruf.
- Über die Funktion *ToClippboard* können Sie eine Palette in die Zwischenablage bewegen. *ToClipboard* erwartet als Parameter eine Referenz auf ein *TClipboard*-Objekt. Beachten Sie, daß die Funktion *ToClipboard* das Objekt von Ihrer Anwendung tatsächlich entfernt. Daher sollten Sie für ein korrektes Funktionieren den Konstruktor von *TPalette* verwenden, um ein temporäres Objekt zu erzeugen:

```
TClipboard clipBoard;
TPalette (tmpPalette).ToClipboard(clipBoard);
```

Erweitern von TPalette

TPalette enthält zwei Funktionen mit geschütztem Zugriff. Beide heißen *Create*. Die beiden Funktionen unterscheiden sich darin, daß eine ein BITMAPINFO * und die andere ein BITMAPCOREINFO * als ersten Parameter erwartet. Diese Funktionen werden vom *TPalette*-Konstruktoren aufgerufen, die ein BITMAPINFO *, ein BITMAPCOREINFO * oder ein *TDib* & erwarten. Die BITMAPINFO * - und BITMAPCOREINFO * -Konstruktoren rufen die korrespondierenden *Create*-Funktionen auf. Der *TDib* & -Konstruktor extrahiert ein BITMAPCOREINFO * oder ein BITMAPINFO * von seinem *TDib*-Objekt und ruft die passende *Create*-Funktion auf.

Beide *Create*-Funktionen erwarten ein UINT für ihren zweiten Parameter. Dieser Parameter ist zum *peFlags*-Element der PALETTEENTRY-Struktur äquivalent und sollte entweder als Null oder mit Werten, die zu *peFlags* kompatibel sind, übergeben werden: PC_EXPLICIT,

PC_NOCOLLAPSE und PC_RESERVED. Ein Paletteneintrag muß das Flag PC_RESERVED gesetzt haben, damit der Eintrag mit der Funktion *AnimatePalette* verwendet werden kann.

Die Funktionen *Create* erzeugen eine LOGPALETTE. Dabei verwenden sie die Farbtabelle aus dem als Parameter übergebenen Bitmap-Header. Sie können *Create* für 2-, 16- und 256-farbige Bitmaps verwenden. Sie scheitert für alle anderen Typen, einschließlich 24-Bit DIBs. Dann benutzt sie die LOGPALETTE, um eine HPALETTE zu erzeugen.

Die Klasse *TBitmap*

Die Klasse *TBitmap* kapselt eine geräteunabhängige Windows-Bitmap und bietet eine Reihe verschiedener Konstruktoren sowie Elementfunktionen für die Manipulation und den Zugriff auf die Bitmap.

Konstruieren von *TBitmap*

Sie können ein *TBitmap*-Objekt entweder direkt mit der Angabe der Breite und Höhe der Bitmap usw. oder indirekt mit der Angabe eines existierenden Bitmap-Objektes, eines Zeigers auf eine BITMAP-Struktur, aus einer Metadatei, aus einem *TDC*-Gerätekontext und anders mehr konstruieren. Hier ist der Konstruktor für das direkte Konstruieren eines Bitmap-Objektes:

```
TBitmap(int width, int height, BYTE planes=1, BYTE count=1, void* bits=0);
```

width und *height* geben die Breite und Höhe der Bitmap in Pixel an. *planes* gibt die Zahl der Farbebenen in der Bitmap an. *count* gibt die Zahl der Bits pro Pixel an. Entweder *plane* oder *count* müssen den Wert eins annehmen. *bits* ist ein Array mit Bits, die in die Bitmap kopiert werden sollen. *bits* kann Null sein. In diesem Falle bleit die Bitmap uninitialisiert.

Sie können Bitmap-Objekte von existierenden Bitmaps, entweder gekapselt in einem *TBitmap*-Objekt oder enthalten in einer BITMAP-Struktur, aus erzeugen.

```
TBitmap(const TBitmap& Bitmap);
TBitmap(const BITMAP far* Bitmap);
```

TBitmap bietet zwei Konstruktoren für das Erzeugen von Bitmap-Objekten, die mit dem gegebenen Gerätekontext kompatibel sind. Der erste Konstruktor erzeugt einfach eine Bitmap der Größe *height* zu *width*. Die Bitmap bleibt uninitialisiert. Wenn Sie beim Parameter *discardable* TRUE angeben, kann die Bitmap gelöscht werden. Eine Bitmap sollte niemals gelöscht werden, wenn sie das aktuell ausgewählte Objekt in einem Gerätekontext ist.

```
TBitmap(const TDC& Dc, int width, int height, BOOL discardable = FALSE);
```

Der zweite Konstruktor erzeugt eine Bitmap, die mit dem Gerät kompatibel ist, welches durch den Gerätekontext aus einer DIB repräsentiert wird. Der Parameter *usage* sollte für 16-Bit-Anwendungen CBM_INIT

sein. CBM_INIT zeigt an, daß die Bitmap mit den Bits aus dem DIB-Objekt initialisiert werden soll. Wenn Sie CBM_INIT nicht angeben, wird die Bitmap zwar erzeugt, jedoch leer gelassen. CBM_INIT ist der Vorgabewert.

32-Bit-Anwendungen können zudem CBM_CREATEDIB angeben. Das Flag CBM_CREATEDIB zeigt an, daß das Farbformat der neuen Bitmap kompatibel mit dem Farbformat aus der BITMAPINFO-Struktur der DIB sein soll. Wenn das Flag CBM_CREATEDIB nicht angegeben ist, wird angenommen daß die Bitmap mit dem gegebenen Gerätekontext kompatibel sein soll.

```
TBitmap(const TDC& Dc, const TDib& dib, DWORD usage);
```

Sie können ferner Bitmaps von der Windows-Zwischenablage, von eine Metadatei oder von einem DIB-Objekt aus erzeugen. Um eine Bitmap von der Zwischenablage aus zu erzeugen, müssen Sie nur eine Referenz auf ein *TClipboard*-Objekt an den Konstruktor übergeben. Der Konstruktor erhält ein Handle der Bitmap in der Zwischenablage und konstruiert ein Bitmap-Objekt aus dem Handle.

```
TBitmap(const TClipboard& clipboard);
```

Um eine Bitmap aus einer Metadatei zu erzeugen, müssen Sie ein *TMetaFilePict* **&**, ein *TPalette* **&** und ein *TSize* **&** übergeben. Der Konstruktor initialisiert eine zum Gerät kompatible Bitmap (basierend auf der Palette) und spielt die Metadatei in die Bitmap.

```
TBitmap(const TMetaFilePict& metaFile, TPalette& palette, const TSize&
size);
```

Um ein Bitmap von einer Geräteunabhängigen Bitmap aus zu erzeugen, müssen Sie ein *TDib* **&** an den Konstruktor übergeben. Sie können zudem eine optionale Palette angeben. Der Konstruktor erzeugt einen Gerätekontext und wandelt die DIB in eine zum Gerät kompatible Bitmap:

```
TBitmap(const TDib& dib, const TPalette* palette = 0);
```

Sie können ein Bitmap-Objekt durch Laden von einem Modul erzeugen. Dieser Konstruktor erwarte zwei Parameter, als erste die HINSTANCE des Moduls, das die Bitmap enthält, und als zweites die Ressource-ID der zu ladenden Bitmap:

```
TBitmap(HINSTANCE, TResId);
```

Sie können ferner ein neues *TBitmap*-Objekt von einem existierenden HBITMAP-Handle aus erzeugen:

```
TBitmap(HBITMAP handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, damit man ein ObjectWindows-Objekt als Alias für ein reguläres Windows-Handle, das in einer Botschaft empfangen wurde, erhält.

Zugriffe auf TBitmap

Sie können auf *TBitmap* über ein HBITMAP oder als eine BITMAP-Struktur zugreifen. Um ein HBITMAP von einem *TBitmap*-Objekt zu erhalten, verwenden Sie den Operator HBITMAP mit dem *TBitmap*-Objekt als Parameter. Der Operator HBITMAP wird fast nie explizit aufgerufen:

```
HBITMAP GetHBitmap(TBitmap &Bitmap)
{
  return Bitmap;
}
```

Dieser Code ruft den Konvertierungsoperator HBITMAP automatisch auf, um das *TBitmap*-Objekt in den korrekten Typ umzuwandeln.

Um ein *TBitmap*-Objekt in eine BITMAP-Struktur umzuwandeln, verwenden Sie die Funktion *GetObject*:

```
BOOL GetBitmap(BITMAP far& dest)
{
  TBitmap Bitmap(200, 100);
  return Bitmap.GetObject(dest);
}
```

Die Funktion *GetObject* füllt nur die Information über Breite, Höhe und Farbformat der BITMAP-Struktur aus. Sie können die aktuellen Bits der Bitmap mit der Funktion *GetBitmapBits* abrufen.

Elementfunktionen

TBitmap kapselt ferner eine Reihe von API-Standardaufrufen für die Manipulation von Paletten:

- Über die folgenden Funktionen können Sie dieselbe Information wie *GetObject* für ein einzelnes Element erhalten. Jede Funktion gibt die Charakteristik des Bitmap-Objektes zurück:

```
int Width();
int Height();
BYTE Planes();
BYTE BitsPixel();
```

- Über die Funktionen *GetBitmapDimension* und *SetBitmapDimension* können Sie die Dimensionen der Bitmap abfragen und ändern. *GetBitmapDimension* erwartet als einzigen Parameter eine Referenz auf ein *TSize*-Objekt. *GetBitmapDimension* legt die Größe der Bitmap in das *TSize*-Objekt ab. *SetBitmapDimension* kann zwei Parameter erwarten, als ersten eine Referenz auf ein *TSize*-Objekt mit der neuen Größe für die Bitmap und als zweites einen Zeiger auf ein *TSize*, in dem die Funktion die alte Größe der Bitmap ablegt. Der zweite Parameter muß nicht an *SetBitmap-*

Dimension übergeben werden. Beide Funktionen geben bei erfolgreicher Operation TRUE zurück.

- Die Funktionen *GetBitmapDimension* und *SetBitmapDimension* betreffen nicht wirklich auf die Größe der Bitmap in Pixeln. Sie ändern vielmehr nur die *physikalische* Größe der Bitmap, die von Programmen oftmals zum Ausdrucken oder Anzeigen einer Bitmap verwendet wird. Dadurch können Sie die Größe der Bitmap abhängig von der Größe der physikalischen Bildschirms anpassen.
- Über die Funktionen *GetBitmapBits* und *SetBitmapBits* können Sie die Bits in einer Bitmap abfragen und ändern. Beider Funktionen erwarten zwei Parameter: ein DWORD und ein **void ***. Der Wert **void *** zeigt auf ein Array und das DWORD legt die Größe des Array in Bytes fest. *GetBitmapBits* füllt das Array bis zu der Zahl von Bytes, die in DWORD angegeben ist, mit Daten aus der Bitmap. *SetBitmapBits* kopiert das Array in die Bitmap. Die Byte-Zahl ist dabei wieder durch DWORD festgelegt.
- Über die Funktion *ToClipboard* können sie eine Bitmap in die Zwischenablage bewegen. *ToClipboard* erwartet eine Referenz auf ein *TClipboard*-Objekt als Parameter. Beachten Sie, daß die Funktion *ToClipboard* das Objekt von Ihrer Anwendung wirklich entfernt. Damit dies keine Fehler verursacht, sollten Sie im allgemeinen über den Konstruktor von *TBitmap* ein temporäres Objekt erzeugen:

```
TClipboard clipBoard;
TBitmap (tmpBitmap).ToClipboard(clipBoard);
```

Erweitern von *TBitmap*

TBitmap hat drei Funktionen mit geschütztem Zugriff, einen Konstruktor und zwei Funktionen mit Namen *Create*.

Der Konstruktor ist ein Standardkonstruktor. Sie können ihn verwenden, wenn Sie eine abgeleitete Klasse konstruieren, um den Konstruktor der Basisklasse nicht explizit aufrufen zu müssen. Wenn Sie den Standardkonstruktor verwenden, müssen Sie die Bitmap in Ihrem eigenen Konstruktor geeignet initialisieren.

Die erste *Create*-Funktion erwartet eine Referenz auf ein *TBitmap*-Objekt als Parameter. Im wesentlichen kopiert diese Funktion das übergebene *TBitmap*-Objekt auf sich selbst.

Die zweite *Create*-Funktion erwartet Referenzen auf ein *TDib*-Objekt und auf ein *TPalette*-Objekt als Parameter. *Create* erzeugt einen Gerätekontext, der mit der *TPalette* kompatibel ist und gibt die DIB in einer zum Gerät kompatiblen Bitmap wieder.

Die Klasse TRegion

Verwenden Sie die Klasse *TRegion*, um einen Bereich in einem Gerätekontext zu definieren. Sie können eine Reihe von Operationen auf einem Gerätekontext durchführen, wobei Sie den Bereich als Schablone benutzen. Dazu zählt beispielsweise das Zeichnen, Füllen, Invertieren usw.. Sie können ferner die Klasse *TRegion* verwenden, um einen Bereich für Ihre eigenen selbstdefinierten Operationen zu definieren.

Konstruieren und Entfernen von TRegion

Bereiche kommen in vielen Formen und Größen vor, von einfachen Rechtecken und Rechtecken mit abgerundeten Ecken bis hin zu komplizierten vieleckigen Formen. Sie können die Form Ihres Bereiches über den verwendeten Konstruktor festlegen. Sie können aber auch einen Bereich indirekt von einem Handle auf einen Bereich oder von einem existierenden *TRegion*-Objekt aus konstruieren.

TRegion bietet einen Standardkonstruktor. Dieser Konstruktor produziert einen leeren rechteckigen Bereich. Mit Hilfe der Funktion *SetRectRgn* können Sie ein leeres *TRegion*-Objekt initialisieren. Nehmen wir zum Beispiel an, Sie leiten Klasse von *TRegion* ab. Rufen Sie dann *SetRectRgn* im Konstruktor Ihrer abgeleiteten Klasse auf, um den Bereich zu initialisieren. Damit müssen Sie den Konstruktor von *TRegion* nicht mehr explizit aufrufen:

```
class TMyRegion : public TRegion
{
public:
  TMyRegion(TRect& rect);
  .
  .
};

TMyRegion::TMyRegion(TRect& rect)
{
  // Initialisiere die Basis TRegion mit rect.
  SetRectRgn(rect);
}
```

Sie können eine *TRegion* von einer Reihe von verschiedenen Quellen aus direkt erzeugen. Zum Erzeugen eines einfachen rechteckigen Bereiches dient der folgende Konstruktor:

```
TRegion(const TRect& rect);
```

Dieser erzeugt ausgehend von den logischen Koordinaten im *TRect*-Objekt einen rechteckigen Bereich.

Zum Erzeugen eines einfachen rechteckigen Bereiches mit abgerundeten Ecken dient der folgende Konstruktor:

```
TRegion(const TRect& rect, const TSize& corner);
```

Dieser erzeugt ausgehend von den logischen Koordinaten im *TRect*-Objekt einen rechteckigen Bereich und rundet die Ecken in eine Ellipse ab. Die Höhe und Breite der verwendeten Ellipse wird durch die Werte im TSize-Objekt definiert.

Zum Erzeugen eines einfachen elliptischen Bereiches dient der folgende Konstruktor:

```
TRegion(const TRect& e, TEllipse);
```

Dieser erzeugt einen durch die logischen Koordinaten in der *TRect*-Struktur begrenzten, elliptischen Bereich. *TEllipse* ist ein Aufzählungswert, der nur einen möglichen Wert annehmen kann, nämlich *Ellipse*. Ein Aufruf dieses Konstruktors sieht in etwa wie folgt aus:

```
TRect rect(20, 20, 80, 60);
TRegion rgn(rect, TRegion::Ellipse);
```

Zum Erzeugen von Bereichen mit einer irregulären mehreckigen Form dient der folgende Konstruktor:

```
TRegion(const TPoint* points, int count, int fillMode);
```

points ist ein Array mit *TPoint*-Objekten. Jedes *TPoint* enthält die logischen Koordinaten einer Ecke des Polygons. *count* gibt die Zahl von Punkten im *points*-Array an. *fillMode* zeigt an, wie der Bereich gefüllt werden sollte; dies kann entweder ALTERNATE oder WINDING sein. Es gibt einen weiteren Konstruktor für das Erzeugen von Bereichen, die aus *mehreren* irregulären mehreckigen Formen besteht:

```
TRegion(const TPoint* points, const int* polyCounts, int count,
        int fillMode);
```

Wie beim anderen Kostruktor für mehreckige Bereiche ist *points* ein Array mit *TPoint*-Objekten. Bei diesem Konstruktor enthält *points* allerdings die Eckpunkte für eine Reihe von Polygonen. *polyCounts* gibt für jedes Polygon die Zahl von Punkten im *points*-Array an. *count* gibt die Gesamtzahl von Polygonen im Bereich und die Zahl von Elementen im *polyCount*-Array an. *fillMode* zeigt an, wie der Bereich gefüllt werden sollte; dies kann entweder ALTERNATE oder WINDING sein.

Nehmen wir zum Beispiel an, Sie konstruieren einen Bereich, der zwei dreieckige Felder umgibt. Jedes Dreieck würde aus drei Punkten bestehen. Damit würde *points* sechs Elemente besitzen, drei für jedes Dreieck. *polyPoints* würde zwei Elemente besitzen, eines für jedes Dreieck. Jedes Element von *polyPoints* würde der Wert drei sein, wodurch die Zahl von Punktion im *points*-Array angegeben würde, die zu jedem Polygon gehört. *count* würde den Wert zwei haben, wodurch angegeben würde, daß der Bereich aus zwei Polygonen besteht.

Sie können eine *TRegion* von einem existierenden HRGN aus erzeugen:

```
TRegion(HRGN handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, damit man ein ObjectWindows-Objekt als Alias für ein reguläres Windows-Handle, das in einer Botschaft empfangen wurde, erhält.

Sie können ferner ein neues *TRegion*-Objekt von einem existierenden *TRegion*-Objekt aus erzeugen:

```
TRegion(const TRegion& region);
```

~TRegion löscht den Bereich und seinen Speicherplatz.

Zugriffe auf TRegion

Sie können auf *TRegion*-Objekte über ein HRGN-Handle oder über eine Reihe von Elementfunktionen und Operatoren direkt zugreifen und sie verändern. Verwenden Sie den Operator HRGN mit dem *TRegion*-Objekt als Parameter, um ein HRGN von einem *TRegion*-Objekt zu erhalten. Der Operator HRGN wird fast nie explizit aufgerufen:

```
HRGN
TMyBitmap::GetHRgn()
{
  return *this;
}
```

Dieser Code ruft automatisch den Konvertierungsoperator HRGN auf, um das *TRegion*-Objekt in den korrekten Typ umzuwandeln.

Elementfunktionen

TRegion bietet eine Reihe von Elementfunktionen, um Information von einem *TRegion*-Objekt zu erhalten. Dazu zählt beispielsweise auch die Frage, ob ein Punkt innerhalb eines Bereiches liegt oder ihn berührt:

- Über die Funktion *SetRectRgn* können Sie den Bereich des Objektes auf ein rechteckiges Feld zurücksetzen:

```
void SetRectRgn(const TRect& rect);
```

Diese Funktion setzt den Bereich von *TRegion* auf die logischen Koordinaten, die in dem als Parameter an die Funktion *SetRectRgn* übergebenen *TRect*-Objekt enthalten sind. Der Bereich wird ungeachtet seiner vorherigen Form auf einen rechteckigen Bereich umgelegt.

- Über die Funktion *Contains* können Sie herausfinden, ob ein Punkt in einem Bereich enthalten ist:

```
BOOL Contains(const TPoint& point);
```

point enthält die Koordinaten des fraglichen Punktes. *Contains* gibt TRUE zurück, wenn *point* innerhalb des Bereiches liegt, anderenfalls FALSE.

- Über die Funktion *Touches* können Sie herausfinden, ob irgendein Teil eines Rechteckes in einem Bereich enthalten ist:

```
BOOL Touches(const TRect& rect);
```

rect enthält die Koordinaten des fraglichen Rechteckes. *Touches* gibt TRUE zurück, wenn irgendein Teil von *rect* innerhalb des Bereiches liegt, und FALSE, wenn nicht.

- Über die Funktionen *GetRgnBox* können Sie die Koordinaten des umgebenden Rechteckes eines Bereiches erhalten:

```
int GetRgnBox(TRect& box);
TRect GetRgnBox();
```

Das umgebende Rechteck ist das kleinste mögliche Rechteck, das alle Felder des Bereiches umfaßt. Die erste Version dieser Funktion erwartet eine Referenz auf ein *TRect*-Objekt als Parameter. Die Funktion legt die Koordinaten des umgebenden Rechtecks im *TRect*-Objekt ab. Der Rückgabewert zeigt die Komplexität des Bereiches an und kann folgende Werte annehmen: SIMPLEREGION (Bereich hat keine überlappende Ränder), COMPLEXREGION (Bereich hat keine überlappende Ränder) oder NULLREGION (Bereich ist leer). Wenn die Funktion fehlschlägt, ist der Rückgabewert ERROR.

Die zweite Version von *GetRgnBox* erwartet keine Parameter und gibt ein *TRect* zurück, das die Koordinaten des umgebenden Rechteckes enthält. Die zweite Version dieser Funktion zeigt nicht die Komplexität dieses Bereiches an.

Operatoren

TRegion besitzt eine große Zahl von Operatoren. Diese Operatoren dienen zur Abfrage und Änderung von Werten in einem Bereich. Sie sind nicht notwendigerweise darauf beschränkt, mit anderen Bereichen zu arbeiten; viele von ihnen ermöglichen Ihnen das Hinzufügen und Abziehen von Rechtecken und anderen Einheiten zum oder vom Bereich.

TRegion besitzt zwei Boolean-Testoperatoren, **==** und **!=**. Diese Operatoren vergleichen zwei Bereiche. Wenn zwei Bereiche äquivalent sind, gibt der Operator **==** TRUE zurück, der Operator **!=** dagegen FALSE. Wenn zwei Bereiche nicht äquivalent sind, gibt der Operator **==** FALSE zurück, der Operator **!=** dagegen TRUE. Sie können diese Operatoren genauso wie deren Äquivalente für **int**s, **char**s usw. verwenden.

Nehmen wir zum Beispiel an, Sie wollen überprüfen, ob zwei Bereiche identisch sind, um einige Operationen auf ihnen ausführen, falls sie es nicht sind. Der Code würde dann in etwa wie folgt aussehen:

```
TRegion rgn1;
TRegion rgn2;
```

```
  // Initialisieren Sie die Bereiche ...

  if(rgn1 != rgn2)
  {
    // Führen Sie Ihre Operationen hier durch
.
.
.

}
```

TRegion bietet ferner eine Reihe von Zuweisungsoperatoren für das Ändern eines Bereiches:

- Mit dem Operator **=** können Sie einen Bereich dem anderen zuweisen. Die Anweisung `rgn1 = rgn2` zum Beispiel setzt den Inhalt von *rgn1* auf den Inhalt von *rgn2*, ungeachtet des Kontextes von *rgn1* vor der Zuweisung.
- Mit dem Operator **+=** können Sie einen Bereich um einen Offset, der in einem *TSize*-Objekt enthalten ist, bewegen. Diese Operation ist analog zu numerischen Additionen: es wird einfach der Offset zu jedem Punkt des Bereiches addiert. Der Bereich behält alle seine Eigenschaften, mit der Ausnahme, daß die Koordinaten, die den Bereich definieren, um die Werte verschoben werden, die in den Elementen *cx* und *cy* des *TSize*-Objektes enthalten sind:

  Ist *cx* positiv, so wird der Bereich um *cx* Pixel nach rechts verschoben.
  Ist *cx* negativ, so wird der Bereich um *cx* Pixel nach links verschoben.
  Ist *cy* positiv, so wird der Bereich um *cy* Pixel nach unten verschoben.
  Ist *cy* negativ, so wird der Bereich um *cy* Pixel nach oben verschoben.

Nehmen wir zum Beispiel an, Sie wollen einen Bereich um 50 Pixel nach rechts und 20 Pixel nach oben bewegen. Der Code würde dann in etwa wie folgt aussehen:

```
TRegion rgn;

// Initialisieren Sie den Bereich...

TSize size(50, -20);
rgn += size;

// Irgendwelche beliebige Operationen mit dem Bereich
.
.
.
```

- Der Operator **–=** führt, wenn er mit einem *TSize*-Objekt benutzt wird, im wesentlichen das Gegenteil des Operators **+=** aus. Das heißt, daß er den Offset einfach von jedem Punkt im Bereich abzieht. Nehmen wir zum Beispiel an, Sie haben denselben Code wie im oben stehenden Beispiel, mit der Ausnahme, daß er den Operator **–=** an Stelle der Operators **+=** verwendet. Dies würde den Bereich in die genau entgegengesetzte Richtung wie beim Operator **+=** verschieben.

- Der mit einem *TRegion*-Objekt benutzte Operator **–=** unterscheidet sich in seinem Verhalten ein wenig von dem mit einem *TSize*-Objekt benutzten Operator **–=**, Betrachten Sie zur Verdeutlichung dieses Unterschiedes folgenden Code:

```
TRegion rgn1, rgn2;

rgn1 -= rgn2;
```

Nach der Ausführung dieses Codes enthält *rgn1* den gesamten Bereich, den er ursprünglich enthielt, minus allen Teilen dieses Bereiches, die auch in *rgn2* enthalten sind. Damit ist nach der Ausführung des Codes jede Punkt, der in *rgn2* enthalten ist, nicht mehr in *rgn1* enthalten. Dies ist analog zur Subtraktion: ziehen Sie den Bereich, definiert durch *rgn2*, von *rgn1* ab.

- Der Operator &= kann sowohl mit *TRegion*-Objekten als auch mit *TRect*-Objekten benutzt werden (bevor irgendwelche Operationen ausgeführt werden, wird *TRect* mit Hilfe des Konstruktors *TRegion::TRegion*(*TRect* &) in ein *TRegion* konvertiert). Betrachten Sie zur Verdeutlichung folgenden Code:

```
TRegion rgn1, rgn2;

rgn1 &= rgn2;
```

Nach der Ausführung dieses Codes enthält *rgn1* den gesamten Bereich, der ursprünglich mit *rgn2* gemeinsam war; das heißt, Gebiete, die vor der Ausführung der Anweisung **&=** in beiden Bereichen vorlagen. Dies ist die Operation eines logischen UND: nur die Gebiete, die Bestandteil von *rgn1* UND *rgn2* waren, werden Bestandteil des neuen Bereiches.

- Der Operator **|=** kann sowohl mit *TRegion*-Objekten als auch mit *TRect*-Objekten benutzt werden (bevor irgendwelche Operationen ausgeführt werden, wird *TRect* mit Hilfe des Konstruktors *TRegion::TRegion*(*TRect* &) in ein *TRegion* konvertiert). Betrachten Sie zur Verdeutlichung folgenden Code:

```
TRegion rgn1, rgn2;

rgn1 |= rgn2;
```

Nach der Ausführung dieses Codes enthält *rgn1* den gesamten Bereich, den er ursprünglich enthielt plus den von *rgn2*; das heißt beide Bereiche. Dies ist die Operation eines logischen ODER: die Gebiete, die Bestandteil von *rgn1* ODER *rgn2* waren, werden Bestandteil des neuen Bereiches.

- Der Operator **^=** kann sowohl mit *TRegion*-Objekten als auch mit *TRect*-Objekten benutzt werden (bevor irgendwelche Operationen ausgeführt werden, wird *TRect* mit Hilfe des Konstruktors *TRegion::TRegion*(*TRect* &) in ein *TRegion* konvertiert). Betrachten Sie zur Verdeutlichung folgenden Code:

```
TRegion rgn1, rgn2;
rgn1 ^= rgn2;
```

Nach der Ausführung dieses Codes enthält *rgn1* nur die Gebiete, die er ursprünglich enthielt und die er nicht mit *rgn2* teilte, plus die Gebiete, die ursprünglich in *rgn2* enthalten waren und die *rgn2* nicht mit *rgn1* teilte. Dieser Operator kombiniert beide Bereiche und entfernt die überlappenden Abschnitte. Dies ist die Operation eines logischen XOR (exklusives ODER): die Gebiete, die Bestandteil von *rgn1* ODER *rgn2*, nicht aber von beiden Bereichen waren, werden Bestandteil des neuen Bereiches.

Die Klasse TIcon

Die Klasse *TIcon* kapselt ein Symbol-Handle sowie Konstruktoren für die Instanziierung eines *TIcon*-Objektes. Sie können die Klasse *TIcon* verwenden, um ein Symbol von einer Ressource oder einer expliziten Info zu konstruieren.

Konstruieren von TIcon

Sie können ein *TIcon* auf eine Reihe unterschiedliche Wege konstruieren: von einem existierenden *TIcon*-Objekt aus, von einer Ressource in der aktuellen Anwendung, von einer Ressource in einem anderen Modul oder explizit von einer Information aus, die die Größe und die Daten enthält.

Sie können Symbolobjekte von einem existierenden Symbol, das in einem *TIcon*-Objekt gekapselt ist, erzeugen:

```
TIcon(HINSTANCE instance, const TIcon& icon);
```

instance kann irgendeine Modulinstanz sein. Sie könnten zum Beispiel die Instanz einer DLL verwenden und das Symbol aus dieser Instanz erhalten:

```
TModule iconLib("MYICONS.DLL");
TIcon icon(iconLib, "MYICON");
```

Beachten Sie die implizite Konvertierung des *TModule iconLib* in ein HINSTANCE im Aufruf des Konstruktors *TIcon*.

Sie können ein *TIcon*-Objekt von einer Symbol-Ressource in irgendeinem Modul erzeugen:

```
TIcon(HINSTANCE instance, TResId resId);
```

In diesem Falle sollte *instance* die HINSTANCE des Moduls sein, von dem Sie das Symbol erhalten möchten, und *resId* ist die Ressource-ID des speziellen gewünschten Symbols. Wenn Sie bei *instance* Null übergeben, erhalten Sie Zugriff auf in Windows eingebaute Symbole.

Sie können ferner ein Symbol von einer Datei laden:

```
TIcon(HINSTANCE instance, char far* filename, int index);
```

In diesem Falle sollte *instance* die Instanz des aktuellen Moduls sein, *filename* ist der Name der Datei, die das Symbol enthält, und *index* ist der Index des Symbols, das wiederhergestellt werden soll.

Sie können ferner ein neues Symbol erzeugen:

```
TIcon(HINSTANCE instance, TSize& size, int planes, int bitsPixel,
      void far* andBits, void far* xorBits);
```

In diesem Falle sollte *instance* die Instanz des aktuellen Moduls sein, *size* gibt die Größe des Symbols an, *planes* die Zahl von Farbebenen, *bitsPixel* die Zahl von Bits pro Pixel, *andBits* zeigt auf ein Array mit einer UND-Maske des Symbols und *xorBits* zeigt auf eine Array mit der XOR-Maske des Symbols. Das Array *andBits* muß eine Monochrommaske angeben. Das Array *xorBits* kann eine monochrome oder eine geräteabhängige Bitmap sein.

Sie können ferner ein neues *TIcon*-Objekt von einem existierenden HICON-Handle aus erzeugen:

```
TIcon(HICON handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, damit man ein ObjectWindows-Objekt als Alias für ein reguläres Windows-Handle, das in einer Botschaft empfangen wurde, erhält.

Es gibt zwei weitere Konstruktoren, die nur für 32-Bit-Anwendungen verfügbar sind:

```
TIcon(const void* resBits, DWORD resSize);
TIcon(const ICONINFO* iconInfo);
```

Der erste Konstruktor erwartet zwei Parameter: *resBits* ist ein Zeiger auf einen Puffer, der die Daten-Bits des Symbols enthält (im allgemeinen durch einen Aufruf der Funktionen *LookupIconIdFromDirectory* oder *LoadResource* erhalten). *resSize* gibt die Zahl der Bits im *resBits*-Puffer an.

Der zweite Konstruktor erwarten als einzigen Parameter, eine ICONINFO-Struktur. Der Konstruktor erzeugt ein Symbol von der Information in der ICONINFO-Struktur. Das Element *fIcon* der Struktur ICONINFO muß TRUE sein, da es damit angibt, daß die ICONINFO-Struktur ein Symbol enthält.

~TIcon löscht das Symbol und seinen Speicherplatz.

Zugriffe auf TIcon

Sie können auf *TIcon* über ein HICON zugreifen. Verwenden Sie den Operator HICON mit dem *TIcon*-Objekt als Parameter, um ein HICON von einem *TIcon*-Objekt zu erhalten. Der Operator HICON wird fast nie explizit aufgerufen:

```
HICON
TMyIcon::GetHIcon()
{
  return *this;
}
```

Dieser Code ruft automatisch den Konvertierungsoperator HICON auf, um das *TIcon*-Objekt in den korrekten Typ umzuwandeln.

Die andere Zugriffsfunktion von *TIcon* ist nur für 32-Bit-Anwendungen verfügbar. Die Funktion hat den Namen *GetIconInfo*. *GetIconInfo* erwartet als einzigen Parameter einen Zeiger auf eine ICONINFO-Struktur. die Funktion füllt die ICONINFO-Struktur aus und gibt TRUE zurück, wenn die Operation erfolgreich war. Nehmen wir zum Beispiel an, Sie erzeugen ein Symbolobjekt und wollen dann die Symboldaten in eine ICONINFO-Struktur extrahieren. Der Code würde dann in etwa wie folgt aussehen:

```
ICONINFO iconInfo;

// Lade vordefiniertes Symbol - Ausrufezeichen
TIcon icon(0, IDI_EXCLAMATION);

icon.GetIconInfo(&iconInfo);
```

Die Klasse TCursor

Die Klasse *TCursor* kapselt ein Cursor-Handle sowie Konstruktoren für das Instanziieren eines *TCursor*-Objektes. Sie können die Klasse *TCursor* verwenden, um einen Cursor von einer Ressource oder einer expliziten Info zu konstruieren.

Konstruieren von TCursor

Sie können einen *TCursor* auf eine Reihe unterschiedliche Wege konstruieren: von einem existierenden *TCursor*-Objekt aus, von einer Ressource in der aktuellen Anwendung, von einer Ressource in einer anderen Anwendung oder explizit von einer Information aus, die die Größe und die Daten enthält.

Sie können Cursor-Objekte von einem existierenden Cursor, das in einem *TCursor*-Objekt gekapselt ist, aus erzeugen:

```
TCursor(HINSTANCE instance, const TCursor& cursor);
```

instance sollte in diesem Falle die Instanz der aktuellen Anwendung sein. *Tcursor* kapselt nicht die Anwendungsinstanz, da *TCursor*-Objekte nichts über Anwendungsobjekte wissen. Normalerweise ist es am einfachsten, auf die aktuelle Anwendungsinstanz in einem Fenster oder anderen Oberflächenobjekten zuzugreifen.

```
TCursor(HINSTANCE instance, TResId resId);


TCursor(HINSTANCE instance, const TPoint& hotSpot,
        TSize& size, void far* andBits, void far* xorBits);
```

Sie können ferner ein neues *TCursor*-Objekt von einem existierenden HCURSOR-Handle aus erzeugen:

```
TCursor(HCURSOR handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, damit man ein ObjectWindows-Objekt als Alias für ein reguläres Windows-Handle, das in einer Botschaft empfangen wurde, erhält.

Es gibt zwei weitere Konstruktoren, die nur für 32-Bit-Anwendungen verfügbar sind:

```
TCursor(const void* resBits, DWORD resSize);
TCursor(const ICONINFO* iconInfo);
```

Der erste Konstruktor erwartet zwei Parameter: *resBits* ist ein Zeiger auf einen Puffer, der die Daten-Bits des Symbols enthält (im allgemeinen durch einen Aufruf der Funktionen *LookupIconIdFromDirectory* oder *LoadResource* erhalten). *resSize* gibt die Zahl der Bits im *resBits*-Puffer an.

Der zweite Konstruktor erwartet als einzigen Parameter, eine ICONINFO-Struktur. Der Konstruktor erzeugt ein Symbol von der Information in der ICONINFO-Struktur. Das Element *fIcon* der Struktur ICONINFO muß FALSE sein, da es damit angibt, daß die ICONINFO-Struktur ein Cursor enthält.

~TIcon löscht den Cursor. Wenn das Löschen fehlschlägt, löst der Destruktor eine Exception aus.

Zugriffe auf TCursor

Sie können auf *TCursor* über ein HCURSOR zugreifen. Verwenden Sie den Operator HCURSOR mit dem *TCursor*-Objekt als Parameter, um ein HCURSOR von einem *TCursor*-Objekt zu erhalten. Der Operator HCURSOR wird fast nie explizit aufgerufen:

```
HCURSOR
TMyCursor::GetHCursor()
{
  return *this;
}
```

Dieser Code ruft automatisch den Konvertierungsoperator HCURSOR auf, um das *TCursor*-Objekt in den korrekten Typ umzuwandeln.

Die andere Zugriffsfunktion von *TIcon* ist nur für 32-Bit-Anwendungen verfügbar. Die Funktion hat den Namen *GetIconInfo*. *GetIconInfo* erwartet als einzigen Parameter einen Zeiger auf eine ICONINFO-Struktur. Die Funktion füllt die ICONINFO-Struktur aus und gibt TRUE zurück, wenn die Operation erfolgreich war. Nehmen wir zum Beispiel an, Sie erzeugen ein Cursorobjekt und wollen dann die Cursordaten in eine ICONINFO-Struktur extrahieren. Der Code würde dann in etwa wie folgt aussehen:

```
ICONINFO cursorInfo;

// Lade vordefinierten Cursor - durchgestrichener Kreis
TCursor cursor(NULL, IDC_NO);

cursor.GetIconInfo(&cursorInfo);
```

Die Klasse TDib

Eine geräteunabhängige Bitmap (oder auch DIB) besitzt kein GDI-Handle wie eine reguläre Bitmap, auch wenn sie ein globales Handle hat. Stattdessen ist sie nur eine Struktur, die Informationen über Format und Palette sowie eine Kollektion von Bits (Pixeln) enthält. Die Klasse *TDib* bietet einen sehr bequemen Weg für einen Umgang mit DIBs wie mit jedem anderen GDI-Objekt. Der Speicher für die DIB liegt in einem Stück, das mit den Windows-Funktionen *GlobalAlloc* reserviert wird. Damit kann sie an die Zwischenablage, an einen OLE-Server oder -Client und an andere Elemente außerhalb der Anwendung, die sie instanziiert hat, übergeben werden.

Konstruieren und Entfernen von TDib

Sie können ein TDib-Objekt von einer Reihe von Quellen aus konstruieren. Dazu zählt das direkte Erzeugen mit der Angabe der Höhe und Breite der Bitmap usw., von der Zwischenablage aus, von einer Ressource aus, von einer Datei oder von einem existierenden DIB-Handle oder *TDib*-Objekt aus. Hier ist der Konstruktor für das direkte Konstruieren eines *TDib*-Objektes:

```
TDib(int width, int height, int nColors, WORD mode=DIB_RGB_COLORS);
```

width und *height* geben die Breite und Höhe der DIB in Pixeln an. *nColors* legt die Zahl der in der DIB benutzten Farben fest. *mode* kann entweder DIB_RGB_COLORS sein, wodurch angegeben wird, daß die Farbtabelle aus RGB-Werten besteht, oder DIB_PAL_COLORS, wodurch angegeben

wird, daß die Farbtabelle aus einem Array mit 16-Bit-Indices auf die aktuell realisierte logische Palette besteht.

Sie können ein *TDib*-Objekt erzeugen, indem Sie es von einem ausführbaren Anwendungsmodul laden. Dieser Konstruktor erwartet zwei Parameter: der erste ist die HINSTANCE des Moduls, das die Bitmap enthält, und der zweite ist die Ressource-ID der zu ladenden Bitmap:

```
TDib(HINSTANCE instance, TResId resId);
```

Um ein *TDib*-Objekt von der Zwischenablage aus zu erzeugen, müssen Sie nur eine Referenz auf ein *TClipboard*-Objekt an den Konstruktor übergeben. Der Konstruktor erhält das Handle der Bitmap in der Zwischenablage und konstruiert ein Bitmap-Objekt aus dem Handle.

```
TDib(const TClipboard& clipboard);
```

Sie können eine DIB aus einer Datei (normalerweise eine .BMP-Datei) in ein *TDib*-Objekt laden, indem Sie den Namen als einzigen Parameter des Konstruktors angeben:

```
TDib(const char* name);
```

Sie können ferner ein *TDib*-Objekt mit einem gegebenen *TBitmap*-Objekt und einem *TPalette*-Objekt konstruieren. Wenn keine Palette gegeben ist, verwendet dieser Konstruktor die aktuell realisierte Palette des fokussierten Fensters.

```
TDib(const TBitmap& Bitmap, const TPalette* pal = 0);
```

Sie können ein DIB-Objekt von einem existierenden DIB-Objekt aus erzeugen:

```
TDib(const TDib& dib);
```

Sie können ferner ein neues *TDib*-Objekt von einem existierenden HGLOBAL-Handle aus erzeugen:

```
TDib(HGLOBAL handle, TAutoDelete autoDelete = NoAutoDelete);
```

Dieser Konstruktor wird verwendet, damit man ein ObjectWindows-Objekt als Alias für ein reguläres Windows-Handle, das in einer Botschaft empfangen wurde, erhält. Da ein HGLOBAL-Handle auf viele verschiedene Arten von Objekten zeigen kann, müssen Sie darauf achten, daß das in diesem Konstruktor verwendete HGLOBAL wirklich das Handle auf eine geräteunabhängige Bitmap ist. Wenn Sie ein Handle auf einen anderen Objekttyp übergeben, löst der Konstruktor eine Exception aus.

Wenn *ShouldDelete* TRUE ist, gibt *~TDib* die Ressource frei und entblockiert das globale Speicherstück und gibt es nach Bedarf frei.

Zugriffe auf *TDib*

TDib bietet eine Reihe verschiedener Typen von Funktionen für den Zugriff auf ein gekapseltes DIB.

Typkonvertierungen

Die Funktionen zur Typkonvertierung für *TDib* ermöglichen Ihnen den Zugriff auf *TDib* in der für die auszuführende Operation bequemsten Art.

Sie können *TDib* über ein HANDLE zugreifen. Verwenden Sie den Operator HANDLE mit dem *TDib*-Objekt als Parameter, um ein HANDLE von einem *TDib*-Objekt zu erhalten. Der Operator HANDLE wird fast nie explizit aufgerufen:

```
HANDLE
TMyDib::GetHandle()
{
  return *this;
}
```

Dieser Code ruft automatisch den Konvertierungsoperator HANDLE auf, um das *TDib*-Objekt in den korrekten Typ umzuwandeln.

Sie können ferner ein *TDib*-Objekt in drei andere Bitmap-Typen konvertieren. Sie können die folgenden Operatoren verwenden, um ein *TDib* in einen dieser Typen zu konvertieren, und dann das Ergebnis überall da benutzen, wo dieser Typ normalerweise verwendet wird:

```
operator BITMAPINFO far*();
operator BITMAPINFOHEADER far*();
operator TRgbQuad far*();
```

Zugriffe auf interne Strukturen

Die Funktionen in diesem Abschnitt geben Ihnen eine Zugriffsmöglichkeit auf die internen Datenstrukturen der DIB. Diese drei Funktionen geben die zur DIB äquivalenten Bitmap-Typen als Zeiger auf BITMAPINFO, BITMAPINFOHEADER und *TRgbQuad*- Objekte zurück:

```
BITMAPINFO far* GetInfo();
BITMAPINFOHEADER far* GetInfoHeader();
TRgbQuad far* GetColors();
```

Die folgende Funktion gibt einen Zeiger auf ein Array mit WORDs zurück, das die Farbindizes für die DIB enthält:

```
WORD far* GetIndices();
```

Die nächste Funktion gibt einen Zeiger auf ein Array mit den Bits, die das aktuelle DIB-Bild ausmachen, zurück:

```
void HUGE* GetBits();
```

Die Zwischenablage

Über die Funktion *ToClipboard* können Sie eine DIB in die Zwischenablage bewegen. *ToClipboard* erwartet als Parameter eine Referenz auf ein

TClipboard-Objekt. Beachten Sie, daß die Funktion *ToClipboard* das Objekt von Ihrer Anwendung tatsächlich entfernt. Damit dies korrekt funktioniert, sollten Sie normalerweise mit dem Konstruktor von *TDib* ein temporäres Objekt erzeugen:

```
TClipboard clipBoard;
TDib (ID_BITMAP).ToClipboard(clipBoard);
```

DIB-Information

Sie können ein *TDib*-Objekt abfragen, um Information über die darin enthaltene DIB zu erhalten.

- Rufen Sie die Funktion *IsOk* auf, um herauszufinden, ob das Objekt gültig ist. *IsOK* erwartet keine Parameter. Sie gibt TRUE zurück, wenn das Objekt gültig ist, und anderenfalls FALSE.
- Die Funktion *IsPM* erwartet keine Parameter. Diese Funktion gibt TRUE zurück, wenn die DIB eine zum Presentation Manager kompatible Bitmap ist.
- Die Funktionen *Width* und *Height* geben die Breite und Höhe der Bitmap in Pixel-Einheiten zurück.
- Die Funktion *Size* gibt die Breite und Höhe der Bitmap in Pixel-Einheiten zurück, allerdings als Bestandteil eines *TSize*-Objektes.
- Die Funktion *NumColors* gibt die Zahl der in der Bitmap verwendeten Farben zurück.
- *StartScan* wurde aus Gründen der Kompatibilität mit älterem Code zur Verfügung gestellt. Diese Funktion gibt immer Null zurück.
- *NumScans* wurde aus Gründen der Kompatibilität mit älterem Code zur Verfügung gestellt. Diese Funktion gibt die Höhe der DIB in Pixeln zurück.
- Die Funktion *Usage* gibt an, in welchem Modus sich die DIB befindet. Dieser Wert ist entweder DIB_RGB_COLORS oder DIB_PAL_COLORS.
- Die Funktion *WriteFile* schreibt das DIB-Objekt auf Platte. Diese Funktion erwartet einen einzelnen Parameter, ein **const char***. Dieser sollte auf den Namen der Datei zeigen, in den die Bitmap gesichert werden soll.

Arbeiten im PAL- oder RGB-Modus

Eine DIB kann Farbwerte aus zwei Arten enthalten. Im Palettenmodus enthält die Farbtabelle der DIB Indizes in eine Palette. Die Farbwerte zeigen nicht selbst eine spezielle Farbe an. Für die Farben müssen erst über die Indizes die korrespondierenden Paletteneinträge in der aktuell realisierten Palette ermittelt werden. Im RGB-Modus repräsentiert jeder Eintrag in der Farbtabelle der DIB einen aktuellen RGB-Farbwert.

Sie können zwischen RGB-Modus und Palettenmodus über die folgenden Funktionen wechseln::

```
BOOL ChangeModeToPal(const TPalette& pal);
BOOL ChangeModeToRGB(const TPalette& pal);
```

Wenn Sie eine Palette über *ChangeModetoPal* umschalten, wird der Parameter *TPalette* **&** als die Palette der DIB benutzt. Jede in der DIB verwendete Farbe wird auf die Palette abgebildet und in einen Palettenindex konvertiert. Wenn Sie mit *ChangeModetoRGB* auf RGB-Modus umschalten, wird der Parameter *TPalette* **&** benutzt, um die aktuellen Palettenindizes in ihre entsprechenden in der Palette enthaltenen RGB-Werte zu konvertieren.

Wenn Sie im RGB-Modus arbeiten, können Sie auf die folgenden Funktionen für den Zugriff und die Änderung der Farbtabelle der DIB zurückgreifen:

- Anfordern eines Eintrages in der Farbtabelle der DIB über die Funktion *GetColor*. Diese Funktion erwartet einen einzelnen Parameter, ein **int**. Er gibt den Index der Farbtabelleneintrages an. *GetColor* gibt ein *TColor*-Objekt zurück.
- Ändern eines Eintrages in der Farbtabelle der DIB über die Funktion *SetColor*. Diese Funktion erwartet zwei Parameter: ein **int** gibt den Index des zu ändernden Farbtabelleneintrages an und ein *TColor* enthält den Wert, auf den der Eintrag geändert werden soll.
- Vergleichen eines *TColor*-Objektes mit den Farbtabelleneinträgen über die Funktion *FindColor*. *FindColor* erwartet einen einzelnen Parameter, ein *TColor*-Objekt. *FindColor* durchsucht die Farbtabelle der DIB, bis eine exakte Übereinstimmung für das *TColor*-Objekt gefunden wurde. Wenn die Suche scheitert, gibt *FindColor* -1 zurück.
- Ersetzen einer Farbe, die aktuell in der Farbtabelle der DIB existiert, durch eine andere Farbe über die Funktion *MapColor*. Diese Funktion erwartet drei Parameter, ein *TColor*-Objekt, das die zu ersetzende Farbe enthält, ein *TColor*-Objekt mit der neuen, in die Farbtabelle einzufügenden Farbe und einen BOOL-Wert, der angibt, ob alle Vorkommnisse der zweiten Farbe ersetzt werden sollen. Wenn der dritte Parameter TRUE ist, werden alle Farbtabelleneinträge, die mit dem ersten Parameter gleich sind, durch den zweiten Parameter ersetzt. Wenn der dritte Parameter FALSE ist, wird nur der erste Farbtabelleneintrag, der mit dem ersten Parameter gleich ist, ersetzt. Standardmäßig ist der dritte Parameter FALSE. Der Rückgabewert dieser Funktion gibt die gesamte Zahl von ersetzten Paletteneinträgen an.

Nehmen wir zum Beispiel an, Sie wollten alle Vorkommnisse von Weiß in Ihrer DIB durch Hellgrau ersetzen. Der Code würde dann in etwa wie folgt aussehen:

```
myDib->MapColor(TColor::LtGray, TColor::White, TRUE);
```

Wenn Sie im Palettenmodus arbeiten, können Sie die folgenden Funktionen verwenden, um auf die Farbtabelle der DIB zuzugreifen und sie zu verändern:

- Anfordern des Eintragsindexes eines beliebigen Farbtabelleneintrages über die Funktion *GetIndex*. Diese Funktion erwartet einen einzelnen Parameter, ein **int**. Er gibt den Index der Farbtabelleneintrages an. *GetIndex* gibt ein WORD zurück, das den Palettenindex enthält.
- Ändern eines beliebigen Eintrages in der Farbtabelle der DIB über die Funktion *SetIndex*. Diese Funktion erwartet zwei Parameter, ein **int** für den Index des zu ändernden Farbtabelleneintrages und ein *WORD* mit dem Palettenindex, auf den hin der Eintrag umgeändert werden soll.
- Vergleichen eines Palettenindexes mit den Farbtabelleneinträgen über die Funktion *FindIndex*. *FindIndex* erwartet einen einzigen Parameter, ein WORD. *FindIndex* durchsucht die Farbtabelle der DIB, bis sie eine Übereinstimmung für das WORD gefunden hat. Wenn die Suche gescheitert ist, gibt *FindIndex* -1 zurück.
- Ersetzen einer aktuell in der Farbtabelle der DIB existierenden Farbe durch eine andere Farbe über die Funktion *MapIndex*. Diese Funktion erwartet drei Parameter: ein WORD für den zu ersetzenden Index, ein WORD für den neuen in die Farbtabelle abzulegenden Palettenindex und ein BOOL-Wert, der angibt, ob alle Vorkommnisse der zweiten Farbe ersetzt werden sollen.

Wenn der dritte Parameter TRUE ist, werden alle Farbtabelleneinträge, die mit dem ersten Parameter gleich sind, durch den zweiten ersetzt. Wenn der dritte Parameter FALSE ist, wird nur der erste Farbtabelleneintrag, der dem ersten Parameter gleicht, ersetzt.

Anpassen von Oberflächenfarben an Systemfarben

DIBs werden oft für das Einrichten einer Benutzeroberfläche verwendet. Damit Ihre Oberfläche mit dem System Ihres Anwendungsbenutzers konsistent wird, sollten Sie die Funktion *MapUIColors* verwenden. Diese Funktion ersetzt die Standardoberflächenfarben durch die benutzereigenen Systemfarben. Hier ist die Syntax für *MapUIColors*:

```
void MapUIColors(UINT mapColors, TColor* bkColor = 0);
```

Der Parameter *mapColors* sollte eine ge-ODER-te Kombination aus fünf Flags sein: *TDib::MapFace*, *TDib::MapText*, *TDib::MapShadow*, *TDib::MapHighlight* und *TDib::MapFrame*. Jeder dieser Werte verursacht eine unterschiedliche Farbersetzung:

Das Flag	ersetzt...	durch...
TDib::MapText	*TColor::Black*	COLOR_BTNTEXT
TDib::MapFace	*TColor::LtGray*	COLOR_BTNFACE

TDib::MapFace	*TColor::Gray*	COLOR_BTNSHADOW
TDib::MapFace	*TColor::White*	COLOR_BTNHIGHLIGHT
TDib::MapFrame	*TColor::LtMagenta*	COLOR_WINDOWFRAME

Falls der Parameter *bkColor* angegeben ist, verursacht er ein Ersetzen der Farbe *TColor::LtYellow* durch die Farbe *bkColor*.

Beachten Sie, daß diese Funktion nur mit einer DIB im RGB-Modus sinnvoll ist, da *MapUIColors* *TColor*-Tabelleneinträge sucht und ersetzt. Da sie ferner spezielle Farben ersetzt, müssen Sie Ihre Oberfläche derart gestalten, daß die korrekten Attribute die geeigneten Farben innehalten: Ihr Schaltertext sollte zum Beispiel schwarz sein (*TColor::Black*), Ihr Schalterhintergrund hellgrau (*TColor::LtGray*), usw.. Dies sollte allerdings kein Problem sein, da diese speziell so entworfen wurden, daß sie zu den Standardvorgabefarben für jedes Oberflächenelement äquivalent sind.

Sie sollten auch die Funktion *MapUIColors* aufrufen, bevor Sie eine mit *MapUIColors* modifizierte Farbe wieder modifizieren. Wenn Sie dies nicht durchführen, kann *MapUIColors* die zu suchenden Attributfarben nicht finden und dieser Teil der Oberfläche wird nicht mit den Systemfarben übereinstimmen.

Erweitern von TDib

TDib bietet eine Reihe geschützter Funktionen, auf die nur innerhalb von *TDib* und von *TDib* abgeleiteten Klassen zugegriffen werden kann. Sie können ebenso auf die Kontrolldaten von *TDib* zugreifen:

- *Info* ist ein Zeiger auf eine BITMAPINFO oder eine BITMAPCOREINFO-Struktur, welche die Attribute, die Farbtabelle und andere Information über die DIB enthält.
- Bits ist ein **void**-Zeiger, der auf einen Speicherbereich zeigt, der die aktuellen grafischen Daten für die DIB enthält.
- NumClrs ist ein **long**-Wert, der die aktuelle Zahl der in der DIB verwendeten Farben enthält. Beachten Sie, daß dies nicht die Zahl aller *möglichen* Farben ist, sondern die Zahl der aktuell benutzten.
- W ist ein **int**-Wert, der die Breite der DIB in Pixeln angibt.
- H ist ein **int**-Wert, der die Höhe der DIB in Pixeln angibt.
- Mode ist ein WORD, das festlegt, ob sich die DIB im RGB-Modus (DIB_RGB_COLORS) oder im Palettenmodus (DIB_PAL_COLORS) befindet.
- IsCore ist ein BOOL, der den Wert TRUE hat, wenn der *Info*-Zeiger auf eine BITMAPCOREINFO-Struktur zeigt. Anderenfalls ist er FALSE.
- IsResHandle gibt an, ob die DIB als eine Ressource geladen wurde und ob deshalb *Handle* ein Ressourcen-Handle ist.

Mit Hilfe der Funktion *InfoFromHandle* können sie die Struktur ausfüllen, auf die durch *Info* gezeigt wird. *InfoFromHandle* extrahiert Information von *Handle* und füllt die Attribute der *Info*-Struktur aus. *InfoFromHandle* erwartet keine Parameter und keinen Rückgabewert.

Die Funktion *Read* liest eine Windows 3.0 DIB oder eine zum Presentation Manager kompatible DIB von einer Datei, die durch ein *TFile*-Objekt referenziert ist. Beim Laden überprüft *Read* den Header, die Attribute die Palette und die Bitmap der DIB. Eine zum Presentation Manager kompatible DIB wird gleichzeitig in eine Windows DIB konvertiert. Diese Funktion gibt TRUE zurück, wenn die DIB korrekt eingelesen wurde.

Mit Hilfe der Funktion *LoadResource* können Sie eine DIB von einer Anwendung oder einem DLL-Modul laden. Diese Funktion erwartet zwei Parameter: eine HINSTANCE, die die Anwendung oder das DLL-Module angibt, von dem Sie die DIB laden wollen, und eine *TResId*, die die spezielle Ressource, die Sie wiederherstellen wollen, innerhalb des Moduls angibt. *LoadResource* gibt TRUE zurück, wenn die Operation erfolgreich war.

Mit Hilfe der Funktion *LoadFile* können Sie eine DIB von einer Datei laden. Diese Funktion erwartet einen Parameter, ein **char ***, der auf einen String mit dem Namen der Datei, die die DIB enthält, zeigt. *LoadFile* gibt TRUE zurück, wenn die Operation erfolgreich war.

K A P I T E L 16

Objekte zur Eingabeprüfung

ObjectWindows gibt Ihnen einen flexiblen Weg an die Hand, um die Information, die ein Benutzer in ein Editierfeld eingibt, zu überprüfen. Dazu werden Prüfungsobjekte mit Editierfeldobjekten assoziiert. Prüfungsobjekte erleichtern das Hinzufügen von Eingabeprüfungen zu existierenden ObjectWindows-Anwendungen oder das Ändern der Art, wie ein Feld seine Daten überprüft.

Dieses Kapitel umfaßt drei Themen, die im Zusammenhang mit der Datenprüfung stehen:

- Die Standardprüfungsklassen
- Verwenden von Datenprüfungsobjekten
- Schreiben von Prüfungen

Prüfungen werden von der Methode *CanClose* von Editierfeldern benutzt. Sie können den Inhalt von jedem einzelnen Editierfeld oder Datenbildschirm zu jeder Zeit überprüfen, indem Sie die Methode *CanClose* aufrufen, die wiederum die Prüfung des Editierfeldes aufruft. Die Prüfungsklassen von ObjectWindows interagieren auch auf der Tastendruckebene und der Ebene, auf der der Fokus zugeteilt und entzogen wird. Die Prüfungsklassen von ObjectWindows automatisieren die Datenprüfung. Meistens werden Sie finden, daß die Datenprüfung nur sehr wenig Mühe erfordert.

Die Standardprüfungsklassen

ObjectWindows umfaßt sechs, in validate.h definierte Prüfungsklassen. Dies sind:

- *TValidator* – eine Basisklasse, von der alle anderen Prüfungsklassen abgeleitet sind.
- TFilterValidator – Eine Filterprüfungsklasse.

- TRangeValidator – eine Prüfungsklasse für numerische Bereiche, die auf *TFilterValidator* basiert.
- TLookupValidator – eine Basisklasse zur Nachschlageprüfung.
- TStringLookupValidator – eine Prüfungsklasse zum Nachschlagen von Strings, die auf *TLookupValidator* basiert.
- TPXPictureValidator – Bildprüfungsklasse, die einen String überprüft, der auf einem gegebenen Muster oder »Bild« basiert.

Die folgenden Abschnitte beschreiben kurz jede der Standardprüfungsklassen.

Prüfungsbasisklassen

Die abstrakte Klasse *TValidator* ist die Basisklasse, von der alle Prüfungsklassen abgeleitet sind. *TValidator* stellt eine Prüfung dar, bei der alle Eingaben gültig sind: die Elementfunktionen *IsValid* und *IsValidInput* liefern immer TRUE zurück und *Error* hat keinen Effekt. Abgeleitete Klassen sollten *IsValid*, *IsValidInput* und *Error* überschreiben, um festzulegen, welche Werte gültig sind und wann Fehler gemeldet werden sollen. Verwenden Sie *TValidator* als Ausgangspunkt für Ihre eigenen Prüfungsklassen, wenn keine der anderen Prüfungsklassen einen passenden Ausgangspunkt darstellen.

Die Filterprüfungsklasse

TFilterValidator stellt eine einfache Prüfung dar, die die Eingabe von Benutzern testet. Der Konstruktor der Filterprüfung erwartet als Parameter eine Menge mit gültigen Zeichen:

TCharSet ist in bitset.h definiert.

```
TFilterValidator(const TCharSet& validChars);
```

TFilterValidator überschreibt *IsValidInput*, um TRUE nur dann zurückzugeben, wenn alle Zeichen im aktuellen Eingabe-String in der Zeichenmenge enthalten sind, die dem Konstruktor übergeben wurde. Das Editierfeld fügt nur Zeichen ein, wenn *IsValidInput* TRUE zurückgibt. Damit müssen Sie *IsValid* nicht überschreiben: Da die Zeichen durch den Eingabefilter gehen, ist der komplette String per Definition gültig. Nachkommen von *TFilterValidator* wie zum Beispiel *TRangeValidator* können die Filterung der Eingabe mit anderen Tests für den vollständigen String kombinieren.

Die Bereichprüfungsklasse

TRangeValidator ist eine Bereichprüfung, die von *TFilterValidator* abgeleitet ist. Sie akzeptiert nur Zahlen und fügt einen Bereichtest für den endgültigen String hinzu. Der Konstruktor erwartet zwei Parameter, die die minimal und maximal gültigen Werte definieren:

```
TRangeValidator(long min, long max);
```

Die Bereichprüfung konstruiert sich selbst als eine Filterprüfung, die nur Ziffern zwischen Null und Neun sowie das Plus- und Minuszeichen akzeptiert Die vererbte Klasse *IsValidInput* stellt damit sicher, daß nur Zah-

len durch den Filter gelangen. *TRangeValidator* überschreibt dann *IsValid*, damit nur dann TRUE zurückgegeben wird, wenn die eingegebenen Zahlen gültige Ganzzahlen innerhalb des im Konstruktor definierten Bereiches sind. Die Methode *Error* zeigt ein Meldungsfenster an, die darauf aufmerksam macht, daß der eingegebene Wert außerhalb des Bereiches liegt.

Die Nachschlageprüfungsklasse

TLookupValidator ist eine abstrakte Klasse, die eingegebene Werte mit eine Liste akzeptierbarer Werte vergleicht, um die Gültigkeit zu bestimmen. *TLookupValidator* führt die virtuelle Elementfunktion *Lookup* ein. Standardmäßig gibt *Lookup* TRUE zurück. Abgeleitete Klassen sollten *Lookup* überschreiben, um das übergebene Element mit einer Liste von Elementen zu vergleichen und TRUE zurückzugeben, wenn eine Übereinstimmung gefunden wurde.

TLookupValidator überschreibt *IsValid*, um nur dann TRUE zurückzugeben, wenn *Lookup* TRUE zurückgibt. In abgeleiteten Klassen sollten Sie *IsValid nicht* und stattdessen eher *Lookup* überschreiben. Die Klasse *TStringLookupValidator* ist eine Instanzklasse, die auf *TLookupValidator* basiert.

Die Nachschlageprüfungsklasse für Strings

TStringLookupValidator ist ein funktionsfähiges Beispiel einer Nachschlageprüfung. Sie vergleicht einen von einem Editierfeld übergebenen String mit den Elementen einer String-Liste. Wenn der übergebene String in der Liste vorkommt, gibt *IsValid* TRUE zurück. Der Konstruktor erwartet als einzigen Parameter die Liste gültiger Strings:

```
TStringLookupValidator(TSortedStringArray* strings);
```

TSortedStringArray ist definiert als:

```
typedef TSArrayAsVector<string> TSortedStringArray;
```

Siehe das Kapitel des *Referenzhandbuches* für eine Beschreibung zu *TSArrayAsVector*.

Um eine andere String-Liste zu verwenden, nachdem Sie die Nachschlageprüfung für Strings konstruiert haben, können Sie die Elementfunktion *NewStringList* verwenden, welche die alte Liste beseitigt und die neue installiert.

TStringLookupValidator überschreibt *Lookup* und *Error*. *Lookup* gibt TRUE zurück, wenn der übergebene String in der Liste vorliegt. *Error* zeigt ein Meldungsfenster an, das darauf aufmerksam macht, daß der String nicht in der Liste vorliegt.

Die Bildprüfungsklasse

Bildprüfungen vergleichen den vom Benutzer eingegebenen String mit einem »Bild« oder Template, das das Formate einer gültigen Eingabe beschreibt. Die verwendeten Bilder sind mit solchen kompatibel, die in der relationalen Datenbank »Paradox« von Borland benutzt werden, um die Benutzereingabe zu kontrollieren. Beim Konstruieren einer Bildprüfung

werden zwei Parameter erwartet: einen String mit dem Template-Bild und einen Boolean-Wert, der angibt, ob das Bild automatisch mit Literalzeichen gefüllt werden soll:

```
TPXPictureValidator(const char far* pic, BOOL autoFill=FALSE);
```

TPXPictureValidator überschreibt *Error*, *IsValidInput* und *IsValid* und fügt zudem eine neue Elementfunktionen *Picture* hinzu. *Error* zeigt ein Meldungsfenster an, das darauf aufmerksam macht, welches Format der String haben sollte. *IsValid* gibt nur dann TRUE zurück, wenn auch die Funktion *Picture* TRUE zurückliefert. Dadurch können Sie neue Bildprüfungen ableiten, indem Sie nur die Elementfunktion *Picture* überschreiben. *IsValidInput* testet Zeichen, wie der Benutzer sie eingibt, und läßt nur solche zu, die nach dem Bildformat erlaubt sind. Optional füllt sie noch Literalzeichen aus dem Bildformat ein.

Hier ist ein Beispiel, wie eine Bildprüfung konstruiert wird:

```
edit->SetValidator(new TPXPictureValidator("###-##-####"));
```

Diese Prüfung ist dazu eingerichtet, um Sozialversicherungsnummern zu akzeptieren. Die Syntax von Bildern ist unter der Elementfunktion *Picture* von *TPXPictureValidator* im *Referenzhandbuch* vollständig beschrieben.

Die Elementfunktion *Picture* versucht, den gegebenen Eingabe-String entsprechend dem Bildformat zu formatieren und gibt einen Wert zurück, der den Grad des Erfolges anzeigt. Der folgende Code listet solche Rückgabewerte auf:

```
//
// Ergebnistyp von TPXPictureValidator
//
enum TPicResult {

  prComplete,
  prIncomplete,
  prEmpty,
  prError,
  prSyntax,
  prAmbiguous,
  prIncompNoFill
};
```

Verwenden von Datenprüfungen

Um Datenprüfungsobjekte zu verwenden, müssen Sie folgendes durchführen:

- Konstruieren Sie ein Editierfeldobjekt.
- Konstruieren Sie ein Prüfungsobjekt und weisen Sie es dem Editierfeld zu.

Wenn Sie einmal eine Prüfung konstruiert und sie mit einem Editierfeld assoziiert haben, müssen Sie nicht mehr mit dem Prüfungsobjekt direkt interagieren. Das Editierfeld weiß, wann Prüfungsmethoden zur richtigen Zeit aufzurufen sind.

Konstruieren eines Editierfeldobjektes

Editierfeldobjekte sind Instanzen der Klasse *TEdit*. Hier ist ein Beispiel dafür, wie ein Editierfeld konstruiert wird:

```
TEdit* edit;
edit = new TEdit(this, 101, sizeof(transfer.NameEdit));
```

Konstruieren und Zuweisen von Prüfungsobjekten

Da Prüfungsobjekte keine Oberflächenobjekte sind, erfordern deren Konstruktoren nur ausreichend Information, um die Prüfungskriterien einzurichten. Ein Prüfungsobjekt für numerische Bereiche erwartet zum Beispiel zwei Parameter: die minimalen und maximalen Werde des gültigen Bereiches.

Jedes Editierfeldobjekt besitzt ein Datenelement, welches auf ein Prüfungsobjekt zeigen kann. Diese Zeigerdeklaration sieht wie folgt aus:

```
TValidator *Validator
```

Wenn *Validator* nicht auf ein Prüfungsobjekt zeigt, verhält sich das Editierfeld wie im Kapitel beschrieben. Sie weisen eine Prüfung zu, indem Sie die Elementfunktion *SetValidator* des Editierfeldobjektes aufrufen. Das Editierfeld testet automatisch mit der Prüfung, wenn es Tastenereignisse verarbeitet oder wenn es aufgerufen wurde, sich selbst zu überprüfen.

Der unten stehende Code zeigt das Konstruieren einer Prüfung und ihre Zuweisung an ein Editierfeld. In diesem Falle erlaubt eine Filterprüfung nur die Verwendung alphabetischer Zeichen.

```
edit->SetValidator(new TFilterValidator("A-Za-z. "));
```

Ein vollständiges Beispiel, das die Verwendung von Standardprüfungen zeigt, ist in \BORLANDC\EXAMPLES\OWL\OWLAPI\VALIDATE zu finden.

Überschreiben von Elementfunktionen einer Prüfung

Die Standardprüfungen sollten für die meisten Datenprüfungsbedürfnisse ausreichen. Sie können aber auch Ihre eigenen Prüfungen schreiben. Dazu müssen Sie mit bestimmten, von der Basisklasse *TValidator* geerb-

ten Elementfunktionen vertraut sein und wissen, wie sie bei Bedarf zu überschreiben sind. Diese Elementfunktionen sind:

- *Valid*
- IsValid
- IsValidInput
- Error

Wenn Sie die Standardprüfungen ändern und Ihre eigenen Prüfungsobjekte schreiben, müssen Sie verstehen, was jede dieser Elementfunktionen ausführt und wie Editierfelder sie verwenden. Alle diese Elementfunktionen werden in den folgenden Abschnitten beschrieben.

Die Elementfunktion *Valid*

Die Elementfunktion *Valid* wird durch das assoziierte Editierfeldobjekt aufgerufen, um die Gültigkeit der eingegebenen Daten zu bestätigen. Vergleichbar mit der Methode *CanClose* von Oberflächenobjekten ist *Valid* eine Boolean-Funktion, die nur dann TRUE zurückgibt, wenn der an sie übergebene String gültig ist. Eine Aufgabe der *CanClose*-Methode eines Editierfeldes ist der Aufruf der *Valid*-Methode des Prüfungsobjektes, an die der aktuelle Text des Editierfeldes übergeben wird.

Wenn Sie Prüfungsobjekte mit Editierfeldern verwenden, sollte für Sie nie ein Aufruf oder Überschreiben der *Valid*-Methode des Prüfungsobjektes nötig sein. Die geerbte Version dieser Elementfunktion wird ausreichen. Standardmäßig gibt *Valid* TRUE zurück, wenn die Methode *IsValid* TRUE zurückgibt; anderenfalls ruft sie *Error* auf, um den Benutzer über einen Fehler zu informieren, und gibt FALSE zurück.

Die Elementfunktion *IsValid*

Die virtuelle Elementfunktion *IsValid* wird durch *Valid* aufgerufen, die an *IsValid* einen zu prüfenden Text-String übergibt. *IsValid* gibt TRUE zurück, wenn der String gültig ist. *IsValid* führt die aktuelle Datenprüfung durch. Wenn Sie deshalb Ihre eigenen Prüfungsobjekte erzeugen, werden Sie wahrscheinlich *IsValid* überschreiben.

Achten Sie darauf, daß Sie *IsValid* nicht direkt aufrufen. Verwenden Sie *Valid* für einen Aufruf von *IsValid*, da *Valid* die Funktion *Error* aufruft, um den Benutzer zu alarmieren, wenn *IsValid* FALSE zurückgibt.

Stellen Sie sicher, daß die Prüfungsaktionen von der Fehlermeldungsaktionen getrennt bleiben.

Die Elementfunktion *IsValidInput*

Wenn ein Editierfeldobjekt ein Tastendruckereignis für sich entdeckt, ruft es die Methode *IsValidInput* des Prüfungsobjektes auf, um sicherzustellen, daß das eingegebene Zeichen eine gültige Eingabe darstellt. Standardmäßig geben die *IsValidInput*-Methoden immer TRUE zurück, was bedeutet, daß alle Tastendrucke akzeptabel sind. Einige abgeleitete Prü-

fungsobjekte überschreiben jedoch *IsValidInput*, um unerwünschte Tastendrucke herauszufiltern.

So geben zum Beispiel Bereichprüfungen, die bei numerischen Eingaben verwendet werden, von *IsValidInput* nur bei numerischen Ziffern und den Zeichen "+" und "-" TRUE zurück.

IsValidInput erwartet zwei Parameter:

```
virtual BOOL IsValidInput(char far* str, BOOL suppressFill);
```

Der erste Parameter *str* zeigt auf den aktuellen Eingabetext, der überprüft wird. Der zweite Parameter ist ein Boolean-Wert, der anzeigt, ob die Prüfung den Eingabe-String zuerst auffüllen soll, bevor er überprüft wird. *TPXPictureValidator* ist das einzige Standardprüfungsobjekt, das den zweiten Parameter benutzt.

Die Elementfunktion *Error*

Die virtuelle Elementfunktion *Error* alarmiert den Benutzer, daß der Inhalt des Editierfeldes die Prüfung nicht erfüllt. Das Standardprüfungsobjekt präsentiert im allgemeinen ein einfaches Meldungsfenster, in dem der Benutzer informiert wird, daß der Inhalt der Eingabe ungültig ist, und in dem beschrieben wird, wie eine geeignete Eingabe aussieht.

Zum Beispiel erzeugt die Elementfunktion *Error* für ein Bereichprüfungsobjekt ein Meldungsfenster, welches angibt, daß der Wert im Editierfeld nicht zwischen dem minimalen und maximalen Wert liegt.

Auch wenn die meisten Prüfungsobjekt-Nachkommen *Error* überschreiben, sollten Sie die Funktion niemals direkt aufrufen. *Valid* ruft *Error* für Sie auf, wenn *IsValid* FALSE zurückgibt. Dies ist das einzige Mal, wo *Error* aufgerufen werden muß.

K A P I T E L 17

Dialogelementobjekte von Visual Basic

Borland C++ 4.5 gibt Ihnen die Möglichkeit, in Ihrer Windows-Anwendung die Dialogelemente von Visual Basic, oder auch VBX-Dialogelemente, zu verwenden. Diese Möglichkeit bezieht sich auf Dialogelemente, die mit Visual Basic 1.0 kompatibel sind. Ihre Verwendung ist genauso einfach wie die eines Standarddialogelemente von Windows oder ObjectWindows.

VBX-Dialogelemente bieten einen großen Umfang an Funktionalität, der bei den Standarddialogelementen von Windows nicht bereitgestellt wird. Ihnen steht ein weites Feld an public-domain-Paketen und kommerziellen Paketen mit VBX-Dialogelementen zur Verfügung, die für eine besser verfeinerte und nützlichere Benutzeroberfläche benutzt werden können.

Dieses Kapitel beschreibt den Entwurf von Anwendungen, die auf VBX-Dialogelemente zurückgreifen. Es erläutert die Klassen *TVbxControl* und *TVbxEventHandler*, geht näher darauf ein, wie Botschaften von einem VBX- Dialogelement zu empfangen sind, und zeigt auf, wie die Eigenschaften eines Dialogelemente abgefragt und gesetzt werden können.

Verwenden von VBX-Dialogelementen

Es gibt eine Reihe von Schritten, die Sie durchführen müssen, um VBX-Dialogelemente in Ihrer ObjectWindows-Anwendung verwenden zu können:

- Rufen Sie in Ihrer Funktion *OwlMain* die Funktion *VBXInit* auf, bevor Sie die Funktion *Run* Ihres Anwendungsobjektes aufrufen. Rufen Sie die Funktion *VBXTerm* auf, nachdem Sie die Funktion *Run* Ihres Anwendungsobjektes aufrufen. *VBXInit* erwartet die Anwendungsinstanz

als ein Parameter. *VBXTerm* hat keine Parameter. Ihrer Funktion *OwlMain* sollte dann etwa wie folgt aussehen:

```
int OwlMain(int argc, char* argv[]) {
  VBXInit(_hInstance);

  TApplication app("Wow!");
  return app.Run();

  VBXTerm();
  }
```

Diese Funktionen initialisieren bzw. schließen die Host-Umgebung jeder Instanz, die für die Verwendung der VBX-Dialogelemente notwendig ist.

- Leiten Sie eine Klasse ab, bei der Sie Ihre Oberlächenbasisklasse mit *TVbxEventHandler* vermischen. Ihre Oberlächenbasisklasse ist eine Klasse, in der Sie das Dialogelement anzeigen wollen. Wenn Sie das Dialogelement in einem Dialogfenster verwenden, müssen Sie *TDialog* zum Mischen verwenden. Der Code würde dann wie folgt aussehen:

```
class MyVbxDialog : public TDialog, public TVbxEventHandler {
public:
  MyVbxDialog(TWindow *parent, char *name)
     : TDialog(parent, name),
       TWindow(parent, name) {}
  DECLARE_RESPONSE_TABLE(MyVbxDialog);
  };
```

- Stellen Sie eine Antworttabelle für das übergeordnete Fenster zusammen, die alle relevanten Ereignisse Ihres Dialogelementes einschließt. Verwenden Sie das Makro EV_VBXEVENTNAME, um die Antwort für jedes Dialogelementereignis einzurichten. Antworttabellen sind in Kapitel 5 detaillierter beschrieben.
- Erzeugen Sie das übergeordnete Fenster des Dialogelementes. Sie können entweder das Dialogelement konstruieren, wenn Sie das übergeordnete Fenster erzeugen, oder vom übergeordneten Fenster selbst aus das Dialogelement konstruieren. Dies hängt davon ab, wie das Dialogelement verwendet wird.

Klassen der VBX-Dialogelemente

ObjectWindows enthält zwei Klassen, die beim Entwurf einer Schnittstelle für VBX-Dialogelemente eingesetzt werden. Diese Klassen sind *TVbxControl* und *TVbxEventHandler*.

Die Klasse TVbxControl

TVbxControl bietet die aktuelle Schnittstelle zum Dialogelement, indem Sie folgendes unterstützt:

- Konstruieren eines VBX-Dialogelementobjektes
- Abfragen und Ändern von Dialogelementeigenschaften
- Herausfinden der Zahl von Dialogelementeigenschaften und Konvertieren von Eigenschaftsnamen von bzw. nach Eigenschaftsindizes
- Herausfinden der Zahl von Dialogelementereignissen und Konvertieren von Ereignisnamen von bzw. nach Ereignisindizes
- Aufrufen der Standardmethoden *AddItem*, *Move*, *Refresh* und *RemoveItem* von Visual Basic 1.0 für Dialogelemente.
- Anfordern des Handle auf das Dialogelement über die Elementfunktion *GetHCTL* von *TVbxControl*

TVbxControl ist von der Klasse *TControl* abgeleitet, die wiederum von *TWindow* abgeleitet ist. Damit agiert *TVbxControl* annähernd auf dieselbe Weise wie jedes andere Oberflächenelement, das auf *TWindow* basiert.

Konstruktoren von TVbxControl

TVbxControl bietet zwei Konstruktoren. Der erste ermöglicht Ihnen ein dynamisches Konstruieren des VBX-Dialogelementes, indem Sie einen Dateinamen, eine Dialogelement-ID, eine Dialogelementklasse, einen Dialogelementtitel, Position und Größe festlegen. Der Konstruktor sieht wie folgt aus:

```
TVbxControl(TWindow *parent,
            int id,
            const char far *FileName,
            const char far *ClassName,
            const char far *title,
            int x, int y,
            int w, int h,
            TModule *module);
```

Dabei gilt folgendes:

- *parent* ist ein Zeiger auf das übergeordnete Fenster des Dialogelementes.
- id ist die Dialogelement-ID. Diese ID wird bei der Definition des Antworttabelle des übergeordneten Fensters benötigt. Im allgemeinen gleicht diese ID sehr stark der Ressource-ID.
- FileName ist der Name der Datei, die das VBX-Dialogelement enthält. Dies kann auch den Pfadnamen einschließen, falls notwendig.
- ClassName ist des Klassenname des Dialogelementes. Eine gegebene Datei für VBX-Dialogelemente kann mehrere separate Dialogelemente beinhalten, die jeweils über einen eindeutigen Klassennamen identifiziert werden. Dieser Name ist für gewöhnlich im Referenzhandbuch für Dialogelemente des dritten Teils der VBX-Dialogelement-Bibliothek zu finden.

- title ist der Titel des Dialogelementes.
- *x* und *y* sind die Koordinaten innerhalb des übergeordneten Fensterobjektes, an denen das Dialogelement plaziert werden soll.
- *w* und *h* sind die Breite und Höhe des Dialogelementes.
- *module* wird an den Basiskonstruktor von *TControl* als der *TModule*-Parameter für diesen Konstruktor übergeben. Der Vorgabewert ist Null.

Der zweite Konstruktor ermöglicht Ihnen, ein *TVbxControl*-Objekt mit einem VBX-Dialogelement einzurichten, welches in der Ressourcendatei der Anwendung definiert wurde. Dieser Konstruktor sieht wie folgt aus:

```
TVbxControl(TWindow *parent,
            TResId resId,
            TModule *module);
```

Dabei gilt folgendes:

- *parent* ist ein Zeiger auf das übergeordnete Fenster des Dialogelementes.
- *resId* ist die Ressource-ID des VBX-Dialogelementes in der Ressourcendatei.
- module wird an den Basiskonstruktor von *TControl* als der *TModule*-Parameter für diesen Konstruktor übergeben. Der Vorgabewert ist Null.

Implizites und explizites Konstruieren

Sie können VBX-Dialogelemente entweder explizit oder implizit konstruieren. Sie konstruieren ein Objekt explizit, wenn Sie einen der Konstruktoren aufrufen. Sie konstruieren ein Objekt implizit, wenn Sie nicht einen der Konstruktoren aufrufen und es zulassen, daß das Dialogelement durch sein übergeordnetes Fenster instanziiert und erzeugt wird.

Explizites Konstruieren beinhaltet den Aufruf eines der Konstruktoren eines VBX-Dialogelementobjektes. Dies wird normalerweise im Konstruktor des übergeordneten Fensters durchgeführt, so daß mit der Erzeugung des übergeordneten Fensters auch das VBX-Dialogelement konstruiert und bereit ist. Sie können aber auch mit dem Konstruieren des Dialogelementes warten, bis es benötigt wird. Eine mögliche Anwendung dafür könnte vorliegen, wenn Sie nur Platz für ein Dialogelement haben. In diesem Falle könnten Sie den Benutzer zur Auswahl in einem Menü oder zum Drücken eines Schalters auffordern. Abhängig von der Benutzerauswahl würden Sie dann ein Objekt instanziieren und in einem existierenden Oberflächenelement anzeigen.

Der folgende Code verdeutlicht das explizite Konstruieren über beide Konstruktoren von *TVbxControl* innerhalb des Konstruktors eines Dialogfensterobjektes:

```
class TTestDialog : public TDialog, public TVbxEventHandler {
public:
```

```
TTestDialog(TWindow *parent, char *name)
   : TDialog(parent, name), TWindow(parent, name) {
   new TVbxControl(this, IDCONTROL1);
   new TVbxControl(this, IDCONTROL2,
                   "SWITCH.VBX", "BiSwitch",
                   "&Program VBX Control",
                   16, 70, 200, 50);
   }
DECLARE_RESPONSE_TABLE(TTestDialog);
};
```

Implizites Konstruieren findet statt, wenn Sie Ihr Oberflächenelement außerhalb des Quelltextes Ihrer Anwendung entwerfen, zum Beispiel im Resource Workshop. Sie können den Resource Workshop verwenden, um VBX-Dialogelemente zu Dialogfenstern und anderen Oberflächenelementen hinzuzufügen. Wenn Sie dann das übergeordnete Fensterobjekt instanziieren, werden die untergeordneten Fenster wie Editierfelder, Listenfelder, Schalter und VBX-Dialogelemente zusammen mit dem übergeordneten Fenster automatisch erzeugt. Auf ähnliche Art werden heutzutage die Dialogfenster oft gemacht. Der unten stehende Code zeigt, wie ein Programm für diese Zwecke aussehen könnte. Hierbei ist allerdings wichtig zu bemerken, daß keines der VBX-Dialogelemente im unten stehenden Code sichtbar ist! Stattdessen ist das VBX-Dialogelement in der Dialog-Ressource DIALOG_1 enthalten. Wenn DIALOG_1 geladen und erzeugt wird, wird das VBX-Dialogelement automatisch erzeugt.

```
class TTestDialog : public TDialog, public TVbxEventHandler {
public:
  TTestDialog(TWindow *parent, char *name)
     : TDialog(parent, name), TWindow(parent, name) {}
  DECLARE_RESPONSE_TABLE(TTestDialog);
  };

void TTestWindow::CmAbout() {
  TTestDialog(this, "DIALOG_1").Execute();
  }
```

Die Klasse TVbxEventHandler

Die Klasse *TVbxEventHandler* ist sehr klein und für die meisten Programmierer nur von geringem Interesse. Ihre Aufgabe ist allerdings sehr wichtig. Ohne die Funktionalität in *TVbxEventHandler* könnten Sie nicht mit Ihren VBX-Dialogelementen kommunizieren. Das Programmiermodell zur Ereignisbehandlung wird weiter unten genauer beschrieben. Dieser Abschnitt erläutert nur den Teil, den *TVbxEventHandler* im Prozeß übernimmt.

TVbxEventHandler besteht aus einer einzigen Funktion und einer Antworttabelle für eine Botschaft. Die Funktion hat den Namen *EvVbxDispatch* und stellt die Routine zur Ereignisbehandlung für eine

Botschaft mit Namen WM_VBXFIREEVENT dar. *EvVbxDispatch* empfängt die Botschaft WM_VBXFIREEVENT, konvertiert die unbehandelte Botschaft zu einer TVBXEVENT-Struktur und sendet eine neue Botschaft ab, die von dem übergeordneten Fenster des Dialogelementes behandelt wird. Da das übergeordnete Fensterobjekt notwendigerweise von *TVbxEventHandler* abgeleitet ist, bedeutet dies, daß das übergeordnete Fenster sich wieder selbst mit einer unterschiedlichen Botschaft aufruft. Die neue Botschaft ist wesentlich einfacher zu behandeln und zu verstehen. Dies ist die Botschaft, die durch das im nächsten Abschnitt beschriebene Makro WM_VBXEVENTNAME behandelt wird.

Behandeln von VBX-Dialogelementbotschaften

Sie müssen die VBX-Dialogelementbotschaften über das zum Dialogelement übergeordnete Fensterobjekt behandeln. Damit das übergeordnete Fensterobjekt zur Behandlung dieser Botschaften in der Lage ist, muß es von der Klasse *TVbxControl* abgeleitet sein. Dazu können Sie eine beliebige Oberflächenobjektklasse, die das VBX-Dialogelement enthalten soll (zum Beispiel *TDialog*, *TFrameWindow* oder Klassen, die Sie von ObjectWindows Oberflächenklassen abgeleitet haben), mit der Klasse *TVbxEventHandler* mischen.

Die Ereignisantworttabelle

Wenn Sie Ihre neue Klasse abgeleitet haben, müssen Sie noch eine Antworttabelle für sie aufbauen. Die Antworttabelle für diese Klasse sieht wie eine normale Antworttabelle aus; Sie müssen nur die regulären Kommandobotschaften und Ereignisse behandeln, die Sie auch normalerweise behandeln würden. Die einzige Erweiterung ist das Makro EV_VBXEVENTNAME, um die neue Klasse von Botschaften von Ihren VBX-Dialogelementen zu behandeln.

Das Makro EV_VBXEVENTNAME erwartet drei Parameter:

```
EV_VBXEVENTNAME(ID, Event, EvHandler)
```

Dabei gilt folgendes:

- *ID* ist die Dialogelement-ID. Sie können diese ID entweder über den zweiten Parameter bei beiden Konstruktoren oder über die Ressource-ID in der Ressourcendatei ermitteln.
- Event ist ein String, der den Ereignisnamen identifiziert. Dieser hängt vom Dialogelement ab und kann einer der Standardnamen für VBX-Ereignisse sein. Sie können diesen Ereignisnamen im Referenzhandbuch für Dialogelemente nachschlagen, sofern das Dialogelement aus dem dritten Teil der VBX-Dialogelement-Bibliothek stammt.

- EvHandler ist die Behandlungsfunktion für dieses Ereignis und Dialogelement. Die Funktion *EvHandler* hat folgenden Kopf:

```
void EvHandler(VBXEVENT FAR *event);
```

Wenn eine Botschaft von ein VBX-Dialogelement durch sein übergeordnetes Fenster empfangen wird, sendet es die Botschaft an die Behandlungsfunktion, die zum korrekten Dialogelement und Ereignis korrespondiert. Wenn es die Funktion aufruft, übergibt es einen Zeiger auf eine VBXEVENT-Struktur. Diese Struktur wird im nächsten Abschnitt genauer beschrieben.

Interpretieren eines Dialogelementereignisses

Wenn ein VBX-Dialogelementereignis eingetreten ist und die Funktion zur Ereignisbehandlung aufgerufen wurde, muß diese Funktion mit der als Parameter empfangenen Struktur TVBXEVENT arbeiten. Diese Struktur sieht wie folgt aus:

```
struct VBXEVENT {
  HCTL    hCtl;
  HWND    hWnd;
  int     nID;
  int     iEvent;

  LPCSTR  lpszEvent;
  int     cParams;
  LPVOID  lpParams;
  };
```

Dabei gilt folgendes:

- *hCtl* ist das Handle des sendenden VBX-Dialogelementes (kein Fenster-Handle).
- hWnd ist das Handle des übergeordneten Fensters.
- nID ist die ID des VBX-Dialogelementes.
- iEvent ist der Ereignisindex.
- lpszEvent ist der Ereignisname.
- cParams ist die Zahl der Parameter für dieses Ereignis.
- lpParams ist ein Zeiger auf ein Array, das Zeiger auf die Parameterwerte für dieses Ereignis enthält

Für ein Verständnis dieser Struktur ist es notwendig zu wissen, wie ein Ereignis eines VBX-Dialogelementes arbeitet. Die ersten drei Elemente der Struktur sind relativ leicht einleuchtend. Sie ermöglichen die Identifizierung des sendenden Dialogelementes. Die nächsten zwei Elemente sind ebenfalls relativ einfach. Jedes Ereignis, das von einem VBX-Dialogelement gesendet werden kann, besitzt sowohl einen Ereignisindex, hier durch *iEvent* repräsentiert, und einen Ereignisnamen, hier durch *lpszEvent* repräsentiert.

Die nächsten beiden Elemente dienen zur Speicherung von Parametern, die mit dem Ereignis übergeben werden. *cParams* enthält die gesamte Zahl von Parametern, die für dieses Ereignis verfügbar sind. *lpParams* ist ein Array mit Zeigern auf die Parameter des Ereignisses (wie bei jedem anderen Array ist auch *lpParam* von Null bis *cParams* - 1 indiziert). Diese Vorgehensweise ist trickreich, da es keinen integrierten Hinweis zu Typ oder Bedeutung jedes Parameters gibt. Stattdessen müssen Sie den Typ von jedem Parameter und die Bedeutung des Parameterwertes kennen. Für die Typen und Bedeutungen der Ereignisparameter können Sie im Referenzhandbuch für Dialogelemente nachschlagen, sofern das Dialogelement aus dem dritten Teil der VBX-Dialogelement-Bibliothek stammt. Anderenfalls müssten Sie das Dialogelement selbst entworfen haben oder Informationen über die Bedeutung der Parameter von der Person erfahren, die das Dialogelement entworfen hat.

Ermitteln von Ereignisinformationen

Im allgemeinen können Sie die von einem Ereignis zurückgegebene Information interpretieren, indem Sie auf die Dokumentation für das VBX-Dialogelement zurückgreifen. Wenn dies nicht ausreicht, bietet *TVbxControl* noch eine Reihe von Mehtoden, um Informationen über ein Ereignis zu erhalten.

Sie können die gesamte Zahl von Ereignissen, die ein Dialogelement senden kann, über die Elementfunktion *GetNumEvents* von *TVbxControl* erfahren. Diese Funktion gibt einen **int**-Wert zurück, der die gesamte Zahl von Ereignissen bestimmt. Diese Ereignisse werden von Null bis zum Rückgabewert von *GetNumEvents* - 1 indiziert.

Sie können den Namen von jedem Ereignis in diesem Bereich erfahren, indem Sie die Elementfunktion *GetEventName* von *TVbxControl* aufrufen. *GetEventName* erwartet einen Parameter, einen **int**-Wert mit dem Index, und gibt einen String zurück, der den Ereignisnamen enthält.

In der anderen Richtung können Sie den Index eines Ereignisses erfragen, indem Sie die Elementfunktion *GetEventIndex* von *TVbxControl* aufrufen. *GetEventIndex* erwartet einen Parameter, einen String mit dem Ereignisnamen, und gibt den korrespondierenden Index als **int**-Wert zurück.

Zugreifen auf ein VBX-Dialogelement

Es gibt zwei Wege, um direkt auf ein VBX-Dialogelement zuzugreifen. Der erste Weg besteht darin, die Eigenschaften des Dialogelementes abzufragen und zu setzen. Ein Dialogelement besitzt eine variable Zahl von Eigenschaften, die Sie festlegen können, um das Aussehen und Verhalten

des Dialogelementes zu beeinflussen. Der andere Weg besteht in dem Aufruf der Methoden des Dialogelementes. Methoden eines Dialogelementes sind den Elementfunktionen in einer Klasse ähnlich und auf sie wird durch Elementfunktionen in der Klasse *TVbxControl* automatisch aktuell zugegriffen. Sie können diese Methoden verwenden, um Aufrufe in das Objekt zu starten und dort eine Aktion auszulösen.

VBX-Dialogelementeigenschaften.

Jedes VBX-Dialogelement besitzt eine Zahl von Eigenschaften. Dialogelementeigenschaften beeinflussen Aussehen und Verhalten des Dialogelementes. Die Eigenschaften schließen beispielsweise die Farben für verschiedene Teile des Dialogelementes ein, die Größe und Position des Dialogelementes, den Dialogelementtitel usw.. Diese Eigenschaften können während der Anzeige des Dialogelementes geändert werden, um das Aussehen oder Verhalten des Dialogelementes, seine Position oder Größe, seine Farbe usw. zu beeinflussen. Ein Ändern dieser Eigenschaften stellt normalerweise Ihrer Hauptmethode für die Manipulation eines VBX-Dialogelementes dar.

Jede Eigenschaft zu einem Dialogelement sollte im Referenzhandbuch für Dialogelemente im dritten Teil der VBX-Dialogelement-Bibliothek vollständig dokumentiert sein.

Wenn das Dialogelement nicht aus dem dritten Teil stammt oder Bestandteil eines kommerziellen Dialogelementpaketes ist, dann müssen Sie sich für die Bedingungen oder speziellen Bedeutungen der Dialogelementeigenschaften an diejenige Person wenden, die das Dialogelement entworfen hat. Viele Eigenschaften fungieren oft nur als ein Index für eine Eigenschaft. Ein Beispiel dafür könnten die Hintergrundmuster sein: 0 könnte einfach bedeuten, 1 Kreuzmuster, 2 schwarz usw.. Ohne geeignete Dokumentation oder Information kann die Verwendung der Dialogelementeigenschaften Schwierigkeiten bereiten.

Ermitteln von Eigenschaftsinformationen

Im allgemeinen können Sie die Informationen über die Eigenschaften eines Dialogelementes erhalten, indem Sie auf die Dokumentation für das VBX-Dialogelement zurückgreifen. Wenn dies nicht ausreicht, bietet *TVbxControl* noch eine Reihe von Mehtoden, um Informationen über die Eigenschaften eines Dialogelementes zu erhalten.

Sie können die gesamte Zahl von Eigenschaften für ein Dialogelement über den Aufruf der Elementfunktion *GetNumProps* von *TVbxControl* abfragen. *GetNumProps* gibt einen **int**-Wert zurück, der die gesamte Zahl von Eigenschaften liefert. Diese Eigenschaften werden von Null bis zum Rückgabewert von *GetNumProps* - 1 indiziert.

Sie können den Namen jeder Eigenschaft innerhalb dieses Bereiches über den Aufruf der Elementfunktion *GetPropName* von *TVbxControl* abfragen.

GetPropName erwartet einen Parameter, den Index als **int**-Wert, und gibt einen String zurück, der den Namen der Eigenschaft enthält.

In der anderen Richtung können Sie den Index einer Eigenschaft über den Aufruf der Elementfunktion *GetPropIndex* von *TVbxControl* abfragen. *GetPropIndex* erwartet einen Parameter, einen String mit dem Eigenschaftsnamen, und gibt den korrespondierenden Eigenschaftsindex als **int**-Wert zurück.

Abfragen der Dialogelementeigenschaften

Sie können den Wert einer Dialogelementeigenschaft entweder über ihren Namen oder über ihre Indexnummer erhalten. Zwar mag die Verwendung des Index etwas effizienter sein (es wird kein String-Vergleich benötigt), die Verwendung des Eigenschaftsnamens ist jedoch wesentlich intuitiver. Auf welche Methode Sie zurückgreifen, häng von Ihrem eigenen Geschmack ab.

TVbxControl stellt die Funktion *GetProp* bereit, um die Eigenschaften eines Dialogelementes anzufordern. *GetProp* ist überladen, damit die Eigenschaften über ihren Name oder Index angefordert werden können. Beide Versionen sind ferner überladen, damit die Zahl verschiedener Eigenschaftstypen abgefragt werden kann:

```
// frage Eigenschaften über Index ab
BOOL GetProp(int propIndex, int& value, int arrayIndex = -1);
BOOL GetProp(int propIndex, long& value, int arrayIndex = -1);
BOOL GetProp(int propIndex, HPIC& value, int arrayIndex = -1);
BOOL GetProp(int propIndex, float& value, int arrayIndex = -1);
BOOL GetProp(int propIndex, string& value, int arrayIndex = -1);

// frage Eigenschaften über Namen ab
BOOL GetProp(const char far* name, int& value, int arrayIndex = -1);
BOOL GetProp(const char far* name, long& value, int arrayIndex = -1);
BOOL GetProp(const char far* name, HPIC& value, int arrayIndex = -1);
BOOL GetProp(const char far* name, float& value, int arrayIndex = -1);
BOOL GetProp(const char far* name, string& value, int arrayIndex = -1);
```

In den Versionen, bei denen der erste Parameter ein **int**-Wert ist, legen Sie die Eigenschaft fest, indem Sie den Eigenschaftsindex übergeben. In den Versionen, bei denen der erste Parameter ein **char ***-Wert ist, legen Sie die Eigenschaft fest, indem Sie den Eigenschaftsnamen übergeben.

Anstatt die Funktion *GetProb* mit dem Eigenschaftswert als Rückgabewert zu beenden, ist der zweite Parameter der Funktion eine Referenz auf den Datentyp der Eigenschaft. Erzeugen Sie ein Objekt desselben Typs wie die Eigenschaft und übergeben Sie eine Referenz auf das Objekt in der Funktion *GetProp*. Wenn *GetProp* zurückkehrt, enthält das Objekt den aktuellen Wert der Eigenschaft.

Der dritte Parameter ist der Index einer Array-Eigenschaft, den Sie bereitstellen sollten, wenn dies für Ihr Dialogelement erforderlich ist. Informationen darüber, ob dieser Parameter erforderlich ist und welche Werte erwartet werden finden Sie in der Dokumentation für Ihr VBX-Dialogelement. Die Funktion ignoriert diesen Parameter, wenn -1 überegeben wird.

Setzen von Dialogelementeigenschaften

Sie können den Wert einer Dialogelementeigenschaft entweder über ihren Namen oder über ihre Indexnummer setzen. Zwar mag die Verwendung des Index etwas effizienter sein (es wird kein String-Vergleich benötigt), die Verwendung des Eigenschaftsnamens ist jedoch wesentlich intuitiver. Auf welche Methode Sie zurückgreifen, häng von Ihrem eigenen Geschmack ab.

TVbxControl stellt die Funktion *SetProp* bereit, um die Eigenschaften eines Dialogelementes zu setzen. *SetProp* ist überladen, damit die Eigenschaften über ihren Name oder Index gesetzt werden können. Beide Versionen sind ferner überladen, damit eine Zahl verschiedener Eigenschaftstypen gesetzt werden kann:

```
// setzen der Eigenschaften über den Index
BOOL SetProp(int propIndex, int value, int arrayIndex = -1);
BOOL SetProp(int propIndex, long value, int arrayIndex = -1);
BOOL SetProp(int propIndex, HPIC value, int arrayIndex = -1);
BOOL SetProp(int propIndex, float value, int arrayIndex = -1);
BOOL SetProp(int propIndex, const string& value, int arrayIndex = -1);
BOOL SetProp(int propIndex, const char far* value, int arrayIndex = -1);

// setzen der Eigenschaften über den Namen
BOOL SetProp(const char far* name, int value, int arrayIndex = -1);
BOOL SetProp(const char far* name, long value, int arrayIndex = -1);
BOOL SetProp(const char far* name, HPIC value, int arrayIndex = -1);
BOOL SetProp(const char far* name, float value, int arrayIndex = -1);
BOOL SetProp(const char far* name, const string& value, int arrayIndex =
-1);
BOOL SetProp(const char far* name, const char far* value, int arrayIndex
            = -1);
```

In den Versionen, bei denen der erste Parameter ein **int**-Wert ist, legen Sie die Eigenschaft fest, indem Sie den Eigenschaftsindex übergeben. In den Versionen, bei denen der erste Parameter ein **char** *-Wert ist, legen Sie die Eigenschaft fest, indem Sie den Eigenschaftsnamen übergeben.

Der zweite Parameter ist der Wert, auf den die Eigenschaft gesetzt werden soll.

Der dritte Parameter ist der Index einer Array-Eigenschaft, den Sie bereitstellen sollten, wenn dies für Ihr Dialogelement erforderlich ist. Informationen darüber, ob dieser Parameter erforderlich ist und welche Werte er-

wartet werden finden Sie in der Dokumentation für Ihr VBX-Dialogelement. Die Funktion ignoriert diesen Parameter, wenn -1 überegeben wird.

Beachten Sie, daß die Funktion *SetProp* bei fünf verschiedenen Datentypen, die Sie an *GetProp* übergeben können, *sechs* verschiedene Datentypen anbietet. Der Grund dafür liegt darin, daß die letzten zwei Versionen beide ein **char *** und die ANSI-Klasse *string* verwenden, um einen String zu repräsentieren. Dies ermöglicht Ihnen eine höhere Flexibilität, wenn Sie ein Zeichenkette an ein Dialogelement übergeben. In der Version von *GetProp* wird eine Typumwandlung bereitgestellt, damit ein **char *** letztlich wie ein *string*-Objekt funktionieren kann.

Methoden von VBX-Dialogelementen

Methoden sind Funktionen, die in jedem VBX-Dialogelement enthalten sind, die Sie für den Aufruf in das Dialogelement verwenden können und die damit eine Aktion auslösen. Welche Aktion ausgeführt wird, hängt von der aufgerufenen Methode ab. *TVbxControl* stellt die Kompatibilität mit den Methoden sicher, die den zu Visual Basic 1.0 kompatiblen Dialogelementen enthalten sind:

- Move(int x, int y, int w, int h)
- Refresh()
- AddItem(int index, const char far *item)
- RemoveItem(int index)

Die Funktion *Move* bewegt das Dialogelement zu den Koordinaten *x*, *y* und verändert die Größe des Dialogelementes zu einer Breite von *w* Pixeln und einer Höhe von *h* Pixeln.

Die Funktion *Refresh* aktualisiert den Darstellungsbereich des Dialogelementes.

Die Funktion *AddItem* fügt das Element *item* der Elementliste des Dialogelementes hinzu und gibt dem neuen Element die Indexnummer *index*.

Die Funktion *RemoveItem* entfernt das Element mit der Indexnummer *index*.

K A P I T E L 18

Dynamische Linkbibliotheken von ObjectWindows

Eine dynamische Linkbibliothek (DLL) ist eine Bibliothek aus Funktionen, Daten und Quellen, deren Bezüge zur Laufzeit und nicht während der Kompilierung aufgelöst werden.

Anwendungen mit statisch gelinktem Code verfügen zur Laufzeit über eine eigene Kopie des Codes. Anwendungen dagegen, die Funktionen aus einer DLL verwenden, teilen sich den Funktionscode mit allen anderen Anwendungen, die diese DLL ebenfalls benutzen. Dadurch reduziert sich die Größe der Anwendungen erheblich. Sie können über eine ObjectWindows-DLL (OWL.DLL) zum Beispiel komplexe Fensterstrukturen definieren, die dann von mehreren Anwendungen gleichzeitig genutzt werden können.

In dem vorliegenden Kapitel wird beschrieben, wie ObjectWindows-DLLs entwickelt und eingesetzt werden.

DLL-Funktionen schreiben

Beim Entwickeln von DLL-spezifischen Funktionen, die von einer Anwendung aus aufgerufen werden sollen, sind folgende Dinge zu beachten:

- Da eine 16-Bit-DLL ein anderes Code- und Datensegment als die sie aufrufende Anwendung nutzt, müssen Aufrufe von Funktionen einer 16-Bit-DLL far-Aufrufe sein und als Parameter übergebene Zeiger far-Zeiger sein. Weiterhin müssen die Rückgabewerte far-Zeiger darstellen. Für 32-Bit-DLLs ist dies nicht unbedingt erforderlich. Mit Hilfe des Makros _FAR können Sie Ihren Quellcode zwischen den Plattformen portierbar gestalten.

- Statische Daten in einer 16-Bit-DLL sind für alle aufrufenden Anwendungen global, weil 16-Bit-DLLs nur ein Datensegment besitzen, das sich alle Instanzen der DLL teilen. Globale Daten einer Anwendung sind daher auch allen anderen Anwendungen zugänglich. Wenn die Daten für den Aufrufer einer 16-Bit-DLL privat sein sollen, dann müssen Sie dynamisch Speicherplatz anfordern und die Daten für die Task selbst manipulieren. Bei 32-Bit-DLLs sind statische Daten für alle Prozesse privat.

Eintritts- und Austrittsfunktionen einer DLL

In jeder Windows-DLL müssen die beiden folgenden Funktionen definiert sein: eine Eintrittsfunktion und eine Austrittsfunktion. In 16-Bit-DLLs wird die Eintrittsfunktion *LibMain* genannt und die Austrittsfunktion *WEP* (Windows Exit Procedure). Die Funktion *LibMain* wird von Windows für die erste Anwendung aufgerufen, die die DLL anspricht, und *WEP* wird von Windows aufgerufen, wenn das letzte Instanz-Handle auf die DLL freigegeben wurde.

Bei 32-Bit-DLLs dient die Funktion *DllEntryPoint* sowohl als Eintritts- als auch als Austrittsfunktion. *DllEntryPoint* wird jedesmal aufgerufen, wenn die DLL geladen oder wieder aus dem Hauptspeicher entfernt wird, wenn ein Prozeß auf die DLL zugreift oder sich von der DLL löst und wenn ein Thread innerhalb eines Prozesses erstellt oder wieder gelöscht wird.

Windows ruft die Eintrittsprozedur (*LibMain* oder *DllEntryPoint*) nur einmal auf, wenn die Bibliothek geladen wird. Die Eintrittsprozedur initialisiert die DLL; diese Initialisierung hängt vollständig von der Funktion der jeweiligen DLL ab, umfaßt aber immer folgende Aufgaben:

- Entsperren des Datensegments mit UnlockDaten, wenn dieses als MOVEABLE deklariert ist.
- Setzen globaler Variablen für die DLL, falls welche verwendet werden.

Der Heap muß nicht unbedingt initialisert werden, weil der Startcode der DLL (C0D_x_.OBJ) den lokalen Heap automatisch initialisiert. In den folgenden Abschnitten werden die Eintritts- und die Austrittsfunktionen einer DLL für 16- und 32-Bit-Anwendungen beschrieben.

LibMain

Die Eintrittsprozedur einer 16-Bit-DLL, *LibMain*, ist folgendermaßen definiert:

```
int PASCAL LibMain(HINSTANCE hInstance, WORD wDatenSeg, WORD cbHeapSize,
                   LPSTR lpCmdLine)
```

Folgende Parameter werden übergeben:

LibMain, HINSTANCE, WORD und LPSTR sind in der windows.h definiert.

- *hInstance* ist das Instanz-Handle der DLL.
- *wDatenSeg* ist der Wert des Datensegmentregisters (DS).
- *cbHeapSize* ist die Größe des lokalen Heap, die in der Moduldefinitionsdatei der DLL angegeben wird.
- *lpCmdLine* ist ein far-Zeiger auf die Kommandozeile, die beim Laden der DLL angegeben wurde. Diese Zeile ist in den meisten Fällen leer, weil DLLs normalerweise automatisch und ohne Parameter geladen werden. Es ist jedoch möglich, eine Kommandozeile an eine DLL zu übergeben, wenn diese explizit geladen werden soll.

Der Rückgabewert von *LibMain* ist entweder 1 (erfolgreiche Initialisierung) oder 0 (Initialisierung nicht erfolgreich abgeschlossen). Windows entfernt die DLL aus dem Hauptspeicher, wenn 0 zurückgegeben wurde.

WEP

WEP ist die Austrittsprozedur einer DLL. Windows ruft diese Funktion auf, bevor die DLL aus dem Speicher entfernt wird. *WEP* ist in einer DLL nicht unbedingt erforderlich (weil in den Laufzeitbibliotheken von Borland C++ eine Standardfunktion enthalten ist). Sie kann jedoch vom Entwickler der DLL mitgeliefert werden, um »Aufräumarbeiten« selbst kontrollieren zu können, bevor die DLL aus dem Speicher gelöscht wird. Oft wird *WEP* aufgerufen, wenn eine Anwendung geschlossen wird, damit gültige Optionen beschränkt werden.

Bei Borland C++ muß *WEP* nicht exportiert werden. Der Prototyp der Funktion WEP sieht folgendermaßen aus:

```
int FAR PASCAL WEP (int nParameter)
```

nParameter ist entweder WEP_SYSTEMEXIT oder WEP_FREE_DLL. Ersteres bedeutet, daß das gesamte Windows heruntergefahren wird. Letzteres bewirkt nur das Entfernen dieser DLL. *WEP* liefert 1, wenn die Funktion erfolgreich durchgeführt werden konnte. Windows verwendet diesen Rückgabewert derzeit aber noch nicht.

DllEntryPoint

Der Eintrittspunkt einer 32-Bit-DLL, *DllEntryPoint*, ist folgendermaßen definiert:

```
BOOL WINAPI DllEntryPoint(HINSTANCE hinstDll, DWORD fdwReason,
                          LPVOID lpvReserved)
```

Es folgt die Beschreibung der Parameter:

- *hinstDll* ist das Instanz-Handle der DLL.

DllEntryPoint ist in winbase.h definiert.

- *fdwReason* ist ein Flag, das angibt, warum die DLL aufgerufen wird (entweder durch einen Prozeß oder durch einen Thread). Folgende Werte sind für diese Flags gültig:

DLL_PROCESS_ATTACH
DLL_THREAD_ATTACH
DLL_THREAD_DETACH
DLL_PROCESS_DETACH

- *lpvReserved* enthält weitere Aspekte zur Initialisierung der DLL und zu den Abschlußarbeiten, die durch den Wert von *fdwReason* festgelegt werden.

DLL-Funktionen exportieren

Nach dem Schreiben der DLL-Funktionen müssen diese exportiert werden, damit sie von einer Anwendung aufgerufen werden können. Dazu sind zwei Schritte erforderlich: Die DLL-Funktionen müssen als exportierbare Funktionen kompiliert und danach exportiert werden. Dies kann auf verschiedene Weise erfolgen:

- Wenn Sie eine Funktion mit dem Schlüsselwort **_export** markieren, dann wird sie als exportierbar kompiliert und anschließend exportiert.
- Wenn Sie das Schlüsselwort **_export** zu einer Klassendeklaration hinzufügen, dann wird die gesamte Klasse (Daten- und Funktionselemente) als exportierbar kompiliert und exportiert.
- Wenn Sie eine Funktion nicht mit **_export** markiert haben, sollten Sie den entsprechenden Compilerschalter oder die IDE-Einstellung verwenden, damit die Funktionen als exportierbar kompiliert werden. Führen Sie die Funktion danach im Abschnitt EXPORTS in der Moduldefinitionsdatei (.DEF) auf.

DLL-Funktionen importieren (aufrufen)

Sie rufen eine DLL-Funktion innerhalb einer Anwendung genauso auf wie jede andere in der Anwendung selbst definierte Funktion. Sie müssen die DLL-Funktionen, die von Ihrer Anwendung aufgerufen werden, jedoch importieren.

Um eine DLL-Funktion zu importieren, können Sie:

- einen IMPORTS-Bereich zur Moduldefinitionsdatei (.DEF) der aufrufenden Anwendung (.DEF) hinzufügen und die DLL-Funktion als zu importieren aufführen.
- beim Linken der aufrufenden Anwendung eine Importbibliothek angeben, die Importinformationen für die DLL-Funktion enthält. (Diese Importbibliothek kann mit Hilfe von IMPLIB erzeugt werden).
- die DLL über *LoadLibrary* explizit laden und sich die Funktionsadressen über *GetProcAddress* ausgeben lassen.

Bei der Ausführung der Anwendung müssen sich die Dateien der aufgerufenen DLLs im aktuellen Verzeichnis, im Pfad oder im Hauptverzeichnis oder System-Verzeichnis von Windows befinden; ansonsten kann Ihre Anwendung nicht auf diese DLL-Dateien zugreifen und sie laden.

Gemeinsam genutzte ObjectWindows-Klassen schreiben

Eine Klasseninstanz kann in einer DLL von mehreren Anwendungen genutzt werden. Sie können zum Beispiel den Code zur Definition der Verhaltensweise eines allgemein verwendbaren Dialogs mehreren Anwendungen zur Verfügung stellen, indem Sie eine Dialogklasse in Ihrer DLL definieren. Dazu müssen Sie diese Klasse aus der DLL exportieren und in Ihre Anwendung importieren. Im folgenden Abschnitt wird erläutert, wie solche gemeinsam genutzten Klassen erstellt werden.

Definieren von gemeinsam genutzten Klassen

Wollen Sie gemeinsam genutzte Klassen definieren, dann müssen Sie:

- Diese Klasse entweder als **_export** oder als **_import** bedingt deklarieren.
- Einen Parameter *TModule** an die Fensterkonstruktoren übergeben (nur in bestimmten Situationen).

Wenn Sie eine gemeinsam genutzte Klasse in einer Include-Datei deklarieren, die sowohl in der DLL als auch in der Anwendung, die diese DLL nutzt, eingebunden wird, dann muß diese Klasse mit dem Schlüsselwort **_export** deklariert sein, wenn die DLL kompiliert wird und mit dem Schlüsselwort **_import**, wenn die Anwendung kompiliert wird. Dies kann mit Hilfe der Definition einer Gruppe von Makros erledigt werden, wobei eines dieser Makros auf **_export** gesetzt werden muß, wenn eine DLL erstellt wird, und auf **_import**, wenn die DLL genutzt wird. Zum Beispiel:

```
#if defined(BUILDEXAMPLEDLL)
   #define _EXAMPLECLASS __export
#elif defined (USEEXAMPLEDLL)
   #define _EXAMPLECLASS __import
#else
   #define _EXAMPLECLASS
#endif

class _EXAMPLECLASS TColorControl : public TControl {
  public:
  .
  .
  .
  };
```

Wenn Sie bei der Erstellung der DLL die Konstante BUILDEXAMPLEDLL definieren (zum Beispiel über die Kommandozeile), dann wird _EXAMPLECLASS in **_export** umgewandelt. Dadurch wird die Klasse exportiert und kann von den Anwendungen, die die DLL verwenden, gemeinsam genutzt werden.

Wenn Sie bei der Erstellung der Anwendung, die die DLL nutzen soll, USEEXAMPLEDLL definieren, dann wird _EXAMPLECLASS in **_import** umgewandelt. Dadurch ist der Anwendung bekannt, welcher Objekttyp importiert werden soll.

Das TModul-Objekt

Eine vollständige Beschreibung der Klasse *TModule* finden Sie im *ObjectWindows Referenzhandbuch*.

Eine Instanz von *TModule* dient als objektorientierter Ersatz für eine ObjektWindows-DLL. Die Elementfunktionen von *TModule* können für die Fenster- und Speicherverwaltung sowie zur Fehlerbehandlung eingesetzt werden.

Das folgende Programmbeispiel zeigt die Deklaration und Initialisierung eines *TModule*-Objekts. Dieses Beispiel enthält Bedingungsanweisungen, so daß für 16-Bit- (LibMain) oder 32-Bit-DLLs (*DllEntryPoint*) die gleiche Quelldatei verwendet werden kann.

```
static TModule *ResMod;

#if defined(__WIN32__)

  BOOL WINAPI
  DllEntryPoint(HINSTANCE instance, DWORD /*flag*/, LPVOID)

#else   // !defined(__WIN32__)
  int
  FAR PASCAL
  LibMain(HINSTANCE instance,
    WORD      /*wDatenSeg*/,
    WORD      /*cbHeapSize*/,
    char far* /*cmdLine*/)
#endif
  {
    // Wir verwenden die DLL und möchten ihre Ressourcen
    // nutzen.
    if (!ResMod)
      ResMod = new TModule(0,instance);
    return TRUE;
  }
```

Innerhalb der Eintrittsfunktion wird das *TModule*-Objekt *ResMod* mit dem Instanz-Handle der DLL initialisiert. Eine Ausnahme wird ausgelöst, wenn das Modul nicht geladen ist.

Wenn die DLL weitere Initialisierungs- und Aufräumarbeiten erfordert, dann können Sie diese über die Funktionen *LibMain, DllEntryPoint* oder *WEP* durchführen. Besser ist es, eine Klasse von *TModule* abzuleiten, Datenelemente für die globalen Daten in der DLL zu definieren und die er-

forderlichen Initialisierungs- und Aufräumarbeiten über deren Konstruktor und Destruktor auszuführen.

Nach dem Kompilieren und Linken der DLL müssen Sie über IMPLIB eine Importbibliothek für diese DLL erzeugen. In dieser Importbibliothek werden alle Elementfunktionen aus den gemeinsam genutzten Klassen sowie alle einfachen Funktionen aufgeführt, die exportiert wurden.

ObjectWindows als DLL

Soll ermöglicht werden, daß ObjectWindows-Anwendungen eine einzige Kopie der ObjectWindows-Bibliothek gemeinsam nutzen, dann sollten Sie diese dynamisch in die DLL von ObjectWindows linken. Dazu sind folgende Schritte erforderlich:

- Bei der Kompilierung muß das Makro _OWLDLL in der Kommandozeile oder in der IDE definiert sein.
- Beim Linken muß anstelle der statischen ObjectWindows--Linkbibliothek (also OWLWS.LIB, OWLWM.LIB, OWLWL.LIB oder OWLWF.LIB) die DLL-Importbibliothek von ObjectWindows (OWLWI.LIB für 16-Bit-Anwendungen oder OWLWFI.LIB für 32Bit-Anwendungen) angegeben werden.

Eine ObjectWindows-DLL aus einer Nicht-ObjectWindows-Anwendung aufrufen

Wird ein untergeordnetes Fenster in einer ObjectWindows-DLL erstellt, das übergeordnete Fenster dagegen in einer ObjectWindows-Anwendung, dann unterstützt ObjectWindows die Kommunikation zwischen dem übergeordneten und dem untergeordneten Fenster. Sie können Ihre DLL jedoch auch für die Verwendung mit Anwendungen vorbereiten, die nicht mit ObjectWindows erstellt wurden.

Wird ein untergeordnetes Fenster in einer ObjectWindows-DLL erstellt, das übergeordnete Fenster dagegen in einer Nicht-ObjectWindows-Anwendung, dann muß die Beziehung zwischen übergeordnetem und untergeordnetem Fenster in der ObjectWindows-DLL simuliert werden. Dazu wird in der ObjectWindows-DLL ein Fensterobjekt erstellt, das mit dem übergeordneten Fenster verknüpft ist. Das Handle dieses Fensters wird in einem DLL-Aufruf angegeben.

Im folgenden Programmbeispiel ist die exportierte Funktion *CreateDLLWindow* in einer ObjectWindows-DLL enthalten. Die Funktion kann sowohl von ObjectWindows- als auch von Nicht-ObjectWindows-Anwendungen genutzt werden.

```
BOOL far _export
CreateDLLWindow(HWND parentHWnd)
{
  TWindow* parentAlias = GetWindowPtr(parentHWnd);
    // Prüfen, ob es sich um ein OWL-Fenster handelt
  if (!parentAlias)
    parentAlias = new TWindow(parentHWnd);
    // Wenn nicht, einen Alias erzeugen
  TWindow* window = new TWindow(parentAlias, "Gruß aus der DLL!");
  window->Attr.Style |= WS_POPUPWINDOW | WS_CAPTION | WS_THICKFRAME
                      | WS_MINIMIZEBOX | WS_MAXIMIZEBOX;
  window->Attr.X = 100; window->Attr.Y = 100;
  window->Attr.W = 300; window->Attr.H = 300;
  return window->Create();
}
```

Die Funktion *CreateDLLWindow* bestimmt, ob das durch den Aufruf *GetWindowPtr* übergebene Handle von einem Nicht-ObjectWindow-Fenster stammt. *GetWindowPtr* liefert 0, wenn das Handle eines Nicht-ObjectWindow-Fensters übergeben wurde. Ist der Rückgabewert dieser Funktion 0, dann wird ein *TWindow*-Ersatzobjekt konstruiert, das als übergeordnetes Fenster eingesetzt wird.

Implizites und explizites Laden

Wenn Sie beim Linken einer Anwendung eine .DEF-Datei oder eine Importbibliothek verwendet haben, dann wird implizit geladen. Die DLL wird durch Windows geladen, sobald eine diese DLL benötigende Anwendung geladen wird.

Explizites Laden erfolgt, wenn DLLs zur Laufzeit geladen werden müssen. Dieser Vorgang erfordert die Windows-API-Funktionen *LoadLibrary* zum Laden der DLL und *GetProcAddress* zur Rückgabe der DLL-Funktionsadressen.

Statische und dynamische Linkbibliotheken mischen

Die ObjectWindows-Bibliotheken werden mit Hilfe der BIDS-Bibliotheken (Klasse container) aufgebaut, die wiederum mit Hilfe der Laufzeit-Bibliothek von C erzeugt wurden.

Wenn Sie beim Linkprozeß mit der DLL-Version der ObjectWindows-Bibliotheken arbeiten, dann müssen Sie auch die DLL-Version der BIDS- und der Laufzeitbibliotheken verwenden. Das erreichen Sie durch die Definition des Makros _OWLDLL. Hierbei handelt es sich aber nicht um die

einzig mögliche Kombination statischer und dynamischer Linkbibliotheken: In der nachfolgenden Tabelle finden Sie in jeder Zeile eine weitere gültige Kombination statischer und dynamischer Linkbibliotheken.

Tabelle 18.1: Erlaubte Bibliotheks-kombinationen

STATIC	**DLL**
OWL, BIDS, RTL	(keine)
OWL, BIDS	RTL
OWL	BIDS, RTL
(keine)	OWL, BIDS, RTL

K A P I T E L 19

Unterstützung von OLE in Borland C++

Dieser Abschnitt des *ObjectWindows Programmierhandbuchs* beschreibt eine Menge von Klassen für die Erstellung von OLE 2 Anwendungen in C++, genannt ObjectsComponents. Dieses einführende Kapitel beantwortet folgende Fragen:

- Was kann ObjectComponents?
- Was ist OLE?
- Wie sieht OLE aus?
- Was ist ObjectComponents?
- Wie arbeitet ObjectComponents?
- Welche Dokumentationen und Hilfsmittel unterstützen die Verwendung von ObjectComponents?
- Was bedeuten alle diese Begriffe?

Die nachfolgenden Kapitel zeigen, wie man mit ObjectComponents unterschiedliche Arten von Programmen erzeugt.

Was kann ObjectComponents?

ObjectComponents vereinfacht die Programmierung mit OLE. Es unterstützt folgende OLE 2 Fähigkeiten:

- Verknüpfen
- Einbetten
- Direkte Bearbeitung im Containerdokument
- Operation Ziehen und Ablegen
- Operationen der OLE-Zwischenablage

- Speichern von Verbunddokumenten
- Automatisierung
- DLL-Server
- Lokalisierung
- Registrierung

Diese Merkmale werden im Abschnitt »ObjectComponents unterstützte OLE 2 Merkmale« genauer beschrieben.

Durch die Benutzung von ObjectComponents erhält man zusätzlich folgende Vorteile:

- Einen einfachen Upgrade-Pfad, um bereits existierende C++ Anwendungen zu verknüpfen und einzubetten, insbesondere bei der Verwendung von ObjectWindows
- Einfache Automatisierung bereits bestehender C++ Anwendungen, mit oder ohne Benutzung von ObjectWindows
- Implementation von Voreinstellungen der Standardelemente der OLE 2 Benutzeroberfläche, wie z.B. das Dialogfenster *Objekt einfügen* oder *Verknüpfung einfügen*.
- Die Möglichkeit, OLE 2 Anwendungen mit AppExpert zu erzeugen, welches die neuen OLE 2 Klassen generiert und versteht
- Kompatibilität mit anderen OLE-Anwendungen, einschließlich OLE 1 Anwendungen, egal ob diese mit Tools von Borland erzeugt wurden oder nicht
- Bei ObjectWindows Anwendungen, die OLE nicht benutzen, gibt es praktisch keinen Zusatzaufwand zur Laufzeit

Wo soll man anfangen?

Das hängt davon ab, was man wissen und was für Anwendungen man erstellen will. Dieser Abschnitt behandelt Themenbereiche, die von Interesse sein können und beschreibt, wo man die benötigte Information findet.

Schreiben von Anwendungen

Die richtige Startposition hängt davon ab, ob man eine neue Anwendung erzeugen oder eine existierende Anwendung anpassen möchte.

Erzeugen einer neuen Anwendung

Man kann eine vollständige OLE-Anwendung fast sofort durch die Benutzung von AppExpert erzeugen. AppExpert unterstützt alle neuen Merkmale von ObjectComponents. Um eine ganz neue OLE-Anwendung zu erstellen, wählt man einfach im IDE-Menü Projekt/AppExpert aus.

Alle weiteren Informationen zu AppExpert findet man im *Benutzerhandbuch*.

Das Programm AppExpert erzeugt die ObjectWindows Bibliothek. Wenn Sie noch nicht mit ObjectWindows gearbeitet haben, fangen Sie am besten mit dem Handbuch O*bjectWindows Einführung* an.

Umwandeln einer Anwendung in einen OLE-Container

Wo man anfängt, hängt davon ab, ob in der Anwendung ObjectWindows verwendet wird oder nicht, und wenn es verwendet wird, ob außerdem das Modell Dok/View verwendet wird.

Tabelle 19.1: Wie man Containerunterstützung in eine existierende Anwendung einfügt

ObjectWindows?	Dok/View?	Wo man beginnt
Ja	Ja	s. Abschnitt »Benutzung von ObjectWindows mit Dok/View Modell«
Ja	Nein	s. Abschnitt »Benutzung von ObjectWindows ohne Dok/View Modell«
Nein	Nein	s. Abschnitt »Schreiben eines Containers ohne ObjectWindows«

Umwandeln einer Anwendung in einen einbettenden und verknüpfenden Server

Wo man anfängt, hängt davon ab, ob in der Anwendung ObjectWindows verwendet wird oder nicht, und wenn es verwendet wird, ob außerdem das Dok/View-Modell verwendet wird.

Tabelle 19.2: Wie man Server in eine existierende Anwendung einfügt

ObjectWindows?	Dok/View?	Wo man beginnt
Ja	Ja	s. Abschnitt »Benutzung von ObjectWindows mit Dok/View-Modell«
Ja	Nein	s. Abschnitt »Benutzung von ObjectWindows ohne Dok/View-Modell«
Nein	Nein	s. Abschnitt »Schreiben eines Servers ohne ObjectWindows«

Hinzufügen von Automatisierungsunterstützung

Das Hinzufügen von Automatisierungsunterstützung unterscheidet nicht zwischen der Benutzung von ObjectWindows oder des Dok/View-Modells. Informationen zum Erstellen von Automatisierten Servern, Controllern oder Typbibliotheken werden in folgenden Abschnitten behandelt:

- **Automatisierte Server:** »Automatisierung einer Anwendung«
- **Automatisierung eines Clients:** »Erzeugen eines automatisierten Controllers«
- **Typbibliothek:**»Erzeugen einer Typbibliothek«

Weitere nützliche Themen

Hier werden einige Themen aufgelistet, die verschiedenartige OLE-Anwendungen gemeinsam haben. Informationen dazu befinden sich in den angegebenen Abschnitten.

- **DLL-Server:** »Erstellen eines DLL Servers«
- **Lokalisierung:** »Lokalisierung von Symbolnamen«
- **Registrierung:** »Erzeugen von Registrierungstabellen«
- **Compilieren und Verknüpfen:** »Erzeugen von ObjectComponents-Anwendungen«
- **Behandlung von Ausnahmen:** »Behandlung von Ausnahmen in ObjectComponents«

Erlernen von ObjectComponents

Die in diesen Abschnitt aufgezählten Aufgaben sollen dabei helfen, sich in ObjectComponents zurechtzufinden.

- **Verstehen von OLE**

  Eine Einführung in OLE befindet sich in Abschnitt »Was ist OLE«. Illustrationen allgemeiner OLE-Wechselwirkungen auf dem Bildschirm werden im Abschnitt »Wie sieht OLE aus« beschrieben.

- **Überblick über die neuen Klassen**

  Für zusammenfassende Tabellen neuer Klassen und Botschaften in ObjectComponents und ObjectWindows siehe Abschnitt »Benutzung von ObjectComponents«.

- **Verstehen der Arbeitsweise von ObjectComponents**

  Der Abschnitt »Wie ObjectComponents arbeitet« erklärt wie ObjectComponents-Klassen Verknüpfungen zwischen OLE und Ihren eigenen C++ Klassen herstellt.

- **Finden von Beispielprogrammen**

  Eine kurze Beschreibung einiger Beispielprogramme von ObjectComponents in Borland C++ befindet sich Abschnitt »Beispielprogramme«.

- **Finden der richtigen Dokumentation**

  Alle Teile der Dokumentation, die ObjectComponents beschreiben, werden in den Abschnitten »Bücher« und »Online-Hilfe« aufgezählt.

- **Verstehen der Begriffe**

  Definitionen der in der Dokumentation verwendeten Begriffe stehen im »Glossar der OLE-Begriffe«. Durchblättern des Glossars ist außerdem eine Möglichkeit, die Grundbegriffe der Merkmale von ObjectComponents kennenzulernen.

Was ist OLE?

OLE ist eine Abkürzung von »Object Linking and Embedding« (»Objekte verknüpfen und einbetten«). OLE ist eine Erweiterung des Betriebssystems, das Anwendungen einen hohen Grad der Integration ermöglicht. OLE definiert einige Standardoberflächen, so daß jedes OLE-Programm mit anderen OLE-Programmen zusammenwirken kann. Keines dieser Progeramme benötigt dabei eingebaute Informationen über die möglichen Partnerprogramme.

Es gibt zwei Formen der OLE 2 Integration. *Verknüpfen und Einbetten* ermöglicht die Einarbeitung aktueller Daten anderer OLE-Anwendungen in die Dateien einer Anwendung. *Automatisierung* läßt eine Anwendung Befehle an eine andere Anwendung ausgeben.

Verknüpfen und Einbetten

Verknüpfen und *Einbetten* beziehen sich auf den Datentransfer von einem Programm zu einem anderen. Das erste Programm, der sogenannte *Server*, schickt seine Daten an das zweite Programm, den *Container*. So können z.B. Felder eines Arbeitsblattes in ein Textverarbeitungsprogramm übertragen werden. Natürlich benötigt man nicht unbedingt OLE, um Daten von einem Windows Programm in ein anderes zu übertragen. Man kann dies auch mit der Zwischenablage erreichen. Der Unterschied zwischen OLE und der Zwischenablage besteht darin, daß in OLE das Programm, das die Daten erhält, keine Informationen über das Format der Daten im Objekt benötigt. Jede beliebige OLE-Server-Anwendung kann seine Daten an jede beliebige OLE-Container-Anwendung weitergeben. OLE ist es zu verdanken, daß es dem Container egal ist, ob die erhaltenen Daten eine Metadatei, eine Bitmap oder ein ASCII-Text sind. Der Server gibt sein intern benutztes Datenformat weiter und der Container akzeptiert es. Außerdem bleibt das Objekt dynamisch, sogar nach der Übertragung. Wenn der Container das Objekt darstellen, verändern oder speichern will, ruft er OLE zur Durchführung auf. Das im Hintergrund arbeitende OLE befiehlt dem Server, den vom Benutzer angegebenen Befehl auszuführen. Aus der Sicht des Benutzers gehört das Objekt zum Container, aber für den Programmierer behält der Server das Objekt und antwortet auf von OLE kommende Befehle. Daraus ergibt sich, daß der Benutzer Objekte aus verschiedenen Servern in einem einzigen Dokument verknüpfen kann, ohne dabei die Möglichkeiten der Server zu verlieren, ihre Objekte zu manipulieren.

Automatisierung

Wir sprechen von Automatisierung, wenn ein Programm Befehle an ein anderes aussendet. Zum Beispiel kann ein Programm für einen Taschenrechner folgende Befehle erhalten:

Drücken der Taste Neun.

Drücken der Taste Plus.

Drücken der Taste Sechs.

Drücken der Taste Gleich.

Ausgabe des Endergebnisses.

Dies sind Befehle, die eine Person normalerweise durch die Benutzeroberfläche des Taschenrechners ausgibt. Durch Automatisierung gibt der Taschenrechner seine internen Befehle für andere Programme frei. Der Taschenrechner wird zum *Automation Object*, und Programme, die ihm Befehle senden, zu *Automation Controllern*. OLE definiert Standardoberflächen, die es einem Controller ermöglichen, jedem installierten Server zu befehlen, eines seiner Objekte zu erzeugen. Des weiteren ermöglicht OLE dem Controller, die Liste der vom Server unterstützten automatisierten Befehle durchzugehen.

Wie sieht OLE aus?

Die verknüpfenden und einbettenden Merkmale von OLE beinhalten eine standardisierte Benutzeroberfläche um allgemeine Operationen durchzuführen. Wie z.B. Plazieren von OLE Objekten in Container-Dateien und deren Aktivierung, sobald sie verknüpft und eingebettet worden sind. OLE standardisiert Befehle des Eingangsmenüs, Dialogfenster, Unterstützung von Ziehen und Ablegen und Painting-Konventionen, so daß die Benutzeroberfläche für OLE-Operationen konsistent über mehrere Anwendungen ist. ObjectComponents und ObjectWindows führen zusammen fast alle Aufgaben der Oberfläche für Sie aus.

Es kann sehr schwer sein, ein OLE-Programm zu verstehen, ohne sich über die zu erzeugende Oberfläche Klarheit verschafft zu haben. Die folgenden Abschnitte werden Bilder eines Containers bei jedem Schritt einer allgemeinen Folge von OLE-Operationen zeigen. Der Benutzer läßt ein Programm laufen, fügt Objekte aus mehreren OLE-Servern in ein Datei ein, editiert ein Objekt und speichert eine Datei. Die Beschreibung dieser Schritte führt Sie in folgende OLE-Merkmale ein:

- Befehl Objekt einfügen
- Befehl Inhalte einfügen
- Einbetten
- Verknüpfen
- Objekt-Aktionsarten
- Auswahl eines Objekts

- Aktivieren eines Objekts
- offenes Bearbeiten
- direktes Bearbeiten im Containerdokument
- Verknüpfen von Menüs

Einfügen eines Objekts

Die erste Abbildung zeigt das Beispielprogramm SdiOle. Es ist ein OLE-Container, der das Single-Document Interface (SDI) benutzt, und wurde mit ObjectComponents und ObjectWindows erstellt. Der Quelltext für SdiOle befindet sich in EXAMPLES/OWL/OCF/SDIOLE.

Das SdiOle-Menü Bearbeiten beinhaltet fünf standardisierte OLE-Befehle, die für die meisten Container gelten: Inhalte einfügen, Verknüpfungen einfügen, Objekt einfügen, Verknüpfungen und Objekt. Die Mehrzahl dieser Befehle sind in der Abbildung nicht auswählbar, da die Zwischenablage leer ist und noch nichts in dem offenen SdiOle-Dokument verknüpft und eingebettet wurde.

Falls dies erwünscht ist, implementiert ObjectWindows alle fünf Standardbefehle, aber ein Container muß nicht unbedingt alle fünf Befehle benutzen. Dieser Abschnitt erklärt nur die Befehle Objekt einfügen und Objekt. Eine kurze Zusammenfassung aller Befehle befindet sich in der Tabelle 19.2.

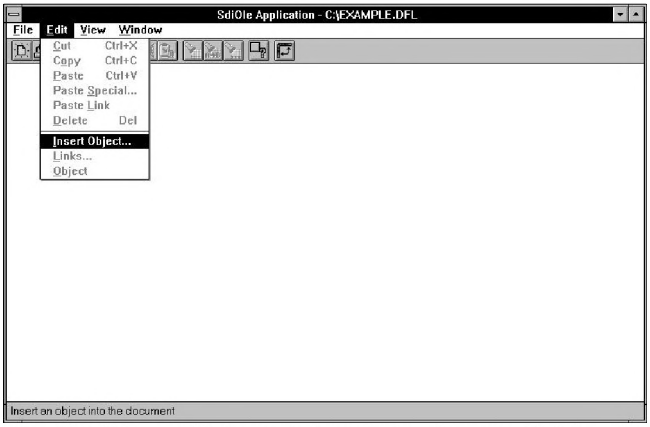


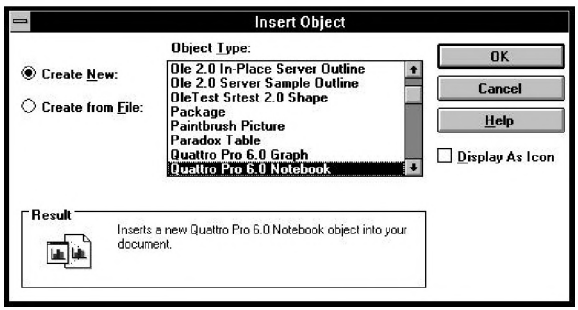
Abbildung 19.1: Das Menü Bearbeiten im Beispielprogramm SdiOle

Die Abbildung 19.2 zeigt das Dialogfenster Objekt einfügen. Genau wie normale Dialogfelder unter Windows, die zum Öffnen von Dateien und zur Auswahl von Schriftarten verwendet werden, so ist das Dialogfenster »Objekte einfügen« eine vom System implementierte Standardressource. Um konsistent zu sein, empfiehlt es sich die Standard-Dialogfenster zu benutzen, außer Ihr Programm enthält so außergewöhnliche Voraussetzungen, daß diese nicht von den Standard-Dialogfenstern erfüllt werden können.

Das Fenster unter Object Type zählt alle Arten der im System erhältlichen Objekte auf. Wenn sich ein Server selbst installiert, dann erklärt er dem System, welche Objekte er erzeugen kann. Das System schreibt diese Informationen in seine Registrierungsdatenbank. Das Dialogfenster Objekt einfügen fragt die Datenbank ab und zeigt alle Typen, die OLE mit der erreichbaren Server-Anwendungen erzeugen kann.

In der Abbildung hat der Benutzer ein Quattro Pro Arbeitsblatt eingefügt. Das Ergebnisfenster im unteren Teil des Dialogfensters erklärt was passiert, wenn der Benutzer jetzt OK anklickt. Da Create New ausgewählt wurde, wird durch klicken auf OK ein neues, leeres Arbeitsblatt in die vom Benutzer geöffnete Datei eingebettet. Abbildung 19.3 zeigt das Ergebnis. (Create from File wird später erklärt werden.)

Abbildung 19.2:
Das Dialogfenster Objekt einfügen



Direkte Bearbeitung eines Objekts im Containerdokument

In der Abbildung 19.3 ist die SdiOle-Anwendung kaum erkennbar. Nur aus dem Titelleiste am oberen Rand des Fensters ist ersichtlich, daß es sich um dieselbe Anwendung handelt. Die Menüleiste hat sich verändert. Sie enthält einige Dropdown-Menüs mehr als in Abbildung 19.1. Die Menüs Block, Arbeitsblatt, Grafiken, Tools und Hilfe sind neu. Es existieren drei Schalterleisten anstelle von einer, und keine von ihnen ähnelt dem Original.

Alle neuen Fensterelemente stammen von Quattro Pro, der Server-Anwendung, die das aktive Objekt erzeugt hat. Quattro Pro hat das SdiOle-Fenster übernommen und stellt seine eigenen Menüs und Mauspaletten dar. Alle Quattro Pro Befehle der Menü- und Mauspalette können hier in SdiOle ausgeführt werden. Das Merkmal von OLE, das es einem Server erlaubt, das Hauptfenster eines Containers zu übernehmen, heißt *direkte Bearbeitung*. Es erlaubt dem Benutzer, das Objekt an Ort und Stelle zu bearbeiten, ohne zwischen den einzelnen Fenstern hin und her schalten zu müssen. Die Programmier-Task, die dies ermöglicht, heißt *Menu Merging* und verknüpft Menüs aus zwei Programmen in einer Menüleiste.

Abbildung 19.3: Ein neu eingefügtes Objekt wird direkt bearbeitet

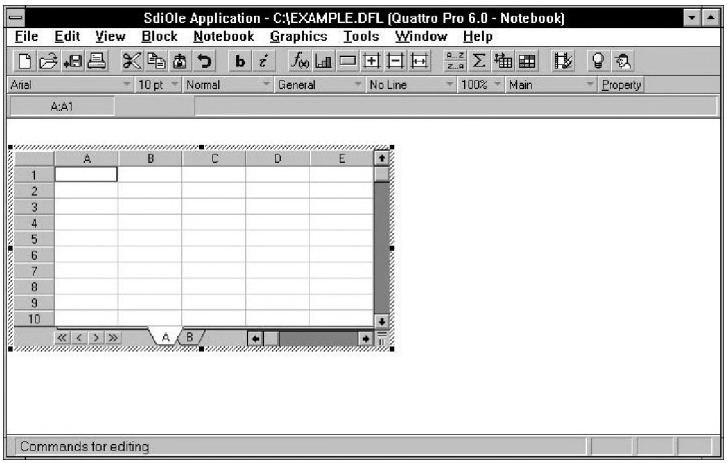
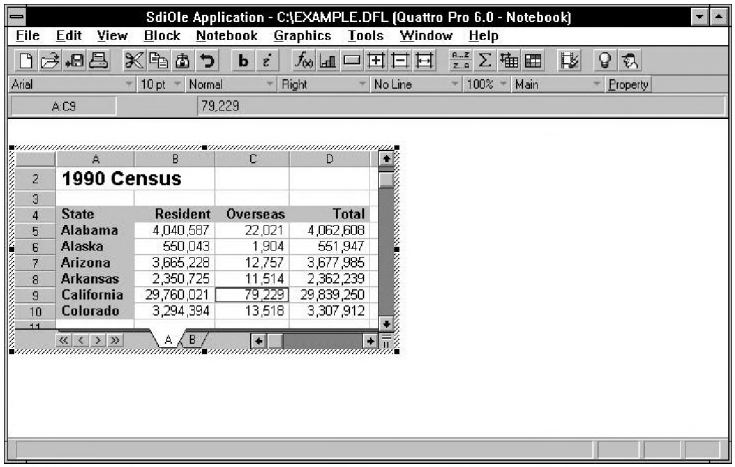


Abbildung 19.4 zeigt, wie das Objekt aussieht nachdem es editiert wurde. Der Benutzer hat Bezeichnungen und Zahlen in Felder eingetragen und hat ein kleines Arbeitsblatt erzeugt. Obwohl manche Programme es zulassen, daß Daten aus anderen Programmen in Dateien eingefügt werden, können ohne OLE die Objekte nicht weiter editiert werden nachdem sie übertragen worden sind.

Abbildung 19.4: Dasselbe Objekt nachdem es editiert wurde



Der Benutzer, der die in Abbildung 19.4 gezeigten Daten eingetragen hat, hatte zweifellos Zugang zu Quattro Pro Hilfmitteln. SdiOle ist eine ganz einfache Anwendung und enthält keine Informationen über Spalten und Reihen oder Schriften und Schattierung. Aber obwohl der Quattro Pro Server das Objekt erzeugt und formatiert hat, gehören die Daten zum Container. Wenn der Benutzer Datei/Speichern aus der Menüleiste auswählt, wird anstelle einer Quattro Pro Datei eine SdiOle-Datei abgespeichert. Mittels ObjectComponents und des OLE-Systems markiert SdiOle

einen Teil seiner eigenen Datei, um die Daten für das eingebettete Objekt abzuspeichern. Wenn der Benutzer Datei/Öffnen auswählt, um dasselbe Objekt wieder zu laden, dann sind die Felder des Arbeitsblattes immer noch vorhanden. Versucht der Benutzer erneut, das Objekt zu bearbeiten, dann ruft OLE Quattro Pro auf, um das SdiOle-Fenster wiederum zu übernehmen. Das Objekt bleibt also mit der Anwendung, die es erzeugt hat verbunden, obwohl das Objekt in einer fremden Datei abgespeichert wurde.

Wenn OLE die Objektdaten so, wie die Daten für dieses Arbeitsblatt direkt in einem Container-Dokument plaziert, dann sagt man, dieses Objekt wurde *eingebettet*. Außer Objekte einzubetten *verknüpft* OLE-Objekte mit Container-Dokumenten, wie in Abbildung 19.7 gezeigt wird.

Aktivieren, Deaktivieren und Auswählen eines Objekts

In den Abbildungen 19.3 und 19.4 wird das eingebettete Objekt durch ein dickes graues Rechteck hervorgehoben. Dieses Rechteck zeigt an, daß das Objekt aktiv ist. Dieses Rechteck erscheint, sobald man ein Objekt mit einem Doppelklick anwählt. Normalerweise wird durch das Aktivieren eines Objekts eine Editiersitzung begonnen, aber der Server entscheidet, ob dieser Konvention gefolgt wird. Zum Beispiel können Ton-Objekte durch Aktivieren abgespielt werden. In den meisten Fällen kann nur ein Objekt auf einmal aktiv sein.

Zum Rechteck in Abbildung 19.4 gehören acht kleine schwarze Boxen, die in regelmäßigen Abständen am äußeren Rand angelegt sind. Sie heißen *Grapples*. Der Benutzer kann durch anklicken eines Grapple und durch ziehen mit der Maus die Größe des Objekts verändern. Außerdem kann durch anklicken an einer beliebigen Stelle des Rechtecks und ziehen mit der Maus das Objekt verschoben werden. ObjektWindows verwendet die Klasse *TUIHandle*, um Rechtecke und Grapples um Objekte zu zeichnen.

In Abbildug 19.5 wird gezeigt, was im Containerfenster passiert, wenn der Benutzer außerhalb des aktivierten Objekts die Maustaste anklickt. Das Rechteck verschwindet. Das Objekt ist jetzt *inaktiv*.

Durch deaktivieren eines Objekts erklärt man OLE, daß man das Bearbeiten beendet hat. Der Server gibt seinen Platz frei, und das Containerfenster nimmt seine normale Gestalt wieder an. Die einzigen in der Menüleiste dargestellten Befehle sind die Befehle, die SdiOle dort plaziert hat. Die Mauspalette und die Fenstertitel haben auch wieder ihr normales Aussehen erhalten.

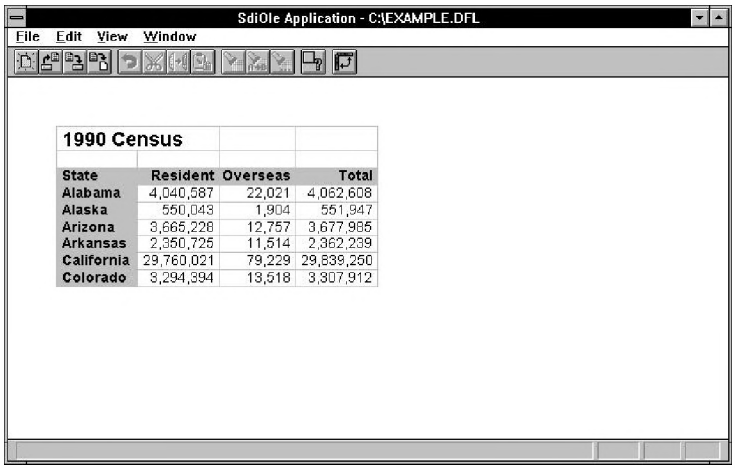


Abbildung 19.5: Die wiederhergestellte Benutzeroberfläche des Containers, nachdem das Objekt deaktiviert wurde

Ein Objekt kann *ausgewählt* werden, ohne dabei aktiviert zu werden. Wenn die Maustaste über einem inaktiven Objekt gedrückt wird, dann zeichnet der Container ein dünnes schwarzes Rechteck um das Objekt. Damit wird angezeigt, daß das Objekt ausgewählt wurde. Dieses Auswahlrechteck ist in Abbildung 19.6 zu sehen. Wie das Aktivierungsrechteck besitzt es Grapples. Ein ausgewähltes Objekt kann genauso wie ein aktives Objekt verändert und verschoben werden.

Auffinden von Objekt-Aktionsarten

Wenn ein Objekt wie in Abbildung 19.6 ausgewählt wurde, dann verändert der Container seine Menüpunkte, um eine Auswahl derjenigen Aktionen anzubieten, die der Server mit dem Objekt ausführen kann. OLE nennt dies Aktionsarten. Üblicherweise bietet der Container benutzbare Aktionsarten an zwei verschiedenen Stellen an: in seinem Menü Bearbeiten und in einem lokalen Menü. Das lokale Menü in Abbildung 19.6 erscheint, wenn das Objekt mit der rechten Maustaste angeklickt wird. Die ersten drei Befehle im lokalen Menü lauten immer Cut (Schneiden), Copy (Kopieren) und Delete (Löschen). Das Erscheinen des vierten Befehls Notebook Objekt (Arbeitsblattobjekt) hängt vom ausgewählten Objekt ab. Wenn z.B. ein Objekt aus Paradox eingefügt wird, dann lautet der vierte Befehl Paradox 5 Object.

Das kleinere überlappende Menü zählt bestimmte vom Server unterstützte Aktionsarten auf. Quattro Pro besitzt nur zwei Aktionsarten. Es kann ein Objekt eröffnen oder ein Objekt bearbeiten. Die Aktionsart Bearbeiten eröffnet eine direkte Bearbeitungssitzung, wie in Abbildung 19.3 gezeigt wird. Die Aktionsart Öffnen eröffnet eine offene Editiersitzung, wie in Abbildung 19.9 dargestellt.

Der letzte Befehl *Convert* (Konvertieren) gilt für alle Objekte. Er ruft ein anderes Standarddialogfenster von OLE auf, das dem Benutzer erlaubt,

das Datenformat eines Servers in ein anderes zu konvertieren. *Convert* erweist sich als sehr nützlich, wenn z.B. Paradox auf einem Rechner installiert wurde, aber es soll ein vollständiges Dokument mit einem eingebetteten Objekt aus einer anderen Datenbankanwendung verarbeitet werden. Wenn Paradox Daten aus einer anderen Datenbank konvertieren kann, dann verknüpft der Befehl *Convert* das fremde Datenbankobjekt mit Paradox.

Abbildung 19.6: Das lokale Menü eines ausgewählten Objekts

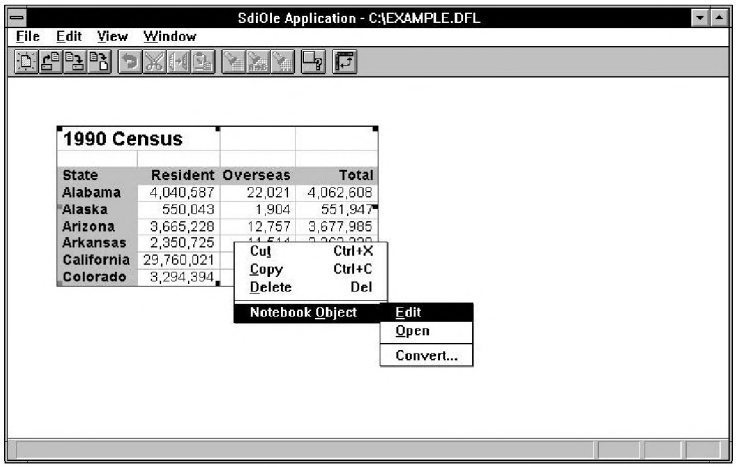


Abbildung 19.8 zeigt, wo Aktionsarten im Menü Bearbeiten auftauchen. Wenn kein Objekt ausgewählt wurde, so ist der letzte Befehl im Menü nicht nutzbar und heißt, wie in Abbildung 19.1, einfach Objekt. Sobald ein Objekt ausgewählt wird, verändert sich der Befehl Objekt, um das ausgewählte Objekt zu beschreiben.

Verknüpfen eines Objekts

Standardmäßig erzeugt der Befehl Objekt einfügen ein ganz neues Objekt, wie in Abbildung 19.3, und bettet dieses ein. Anstatt das Objekt einzubetten, kann man das Objekt auch verknüpfen.

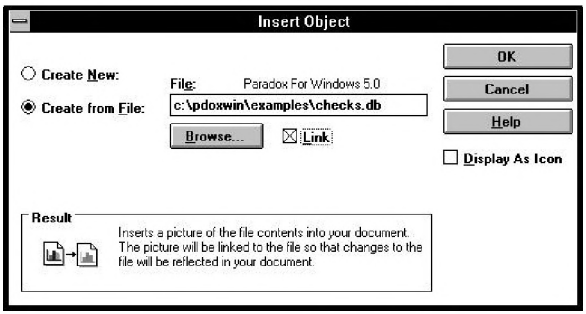
Wenn OLE ein Objekt verknüpft, dann werden die Objektdaten nicht im Dokument des Containers abgespeichert. Es speichert nur den Namen des Servers ab, in dem die Daten gespeichert sind (inklusive der Lage der Daten innerhalb der Datei und einem Bild des Objekts wie es auf dem Bildschirm erscheint. Dieses Bild ist normalerweise eine Metadatei.). Der Container erhält keine Kopie des Objekts, er erhält einen Zeiger zum Objekt. Dennoch zeichnet OLE das Objekt in das Container-Dokument ein, genauso als wäre es eingebettet, aber die Daten gehören nicht zum Container.

Wenn das die Daten für das verbundene Objekt beinhaltende Server-Dokument gelöscht wird, dann kann der Benutzer das verbundene Objekt

nicht mehr aktivieren oder bearbeiten. Andererseits werden die Daten automatisch in allen mit demselben Objekt verbundenen Container-Dokumenten aktualisiert, sobald die Daten im Server-Dokument verändert werden. Wenn mehrere Dokumente dasselbe Objekt einbetten, dann erzeugen sie Kopien, und Änderungen in einem Dokument haben keine Auswirkungen auf die Kopien in anderen Dokumenten.

Abbildung 19.7 zeigt, was geschieht, wenn der Schalter Von Datei erzeugen im Dialogfenster Objekt einfügen ausgewählt wird. Anstatt ein neues leeres Objekt zu erzeugen, kann eine Datei mit bestehenden Daten gewählt werden. OLE ruft dann den Server auf, der diese Datei erstellt hat. Daten aus dieser Datei können eingebettet werden, aber in unserem Beispiel in Abbildung 19.7 wurde das Feld Verknüpfen angekreuzt, d.h. wenn der Benutzer OK anklickt, dann kopiert OLE keine Daten aus CHECKS.DB in das Server-Dokument. Es erzeugt eine Verknüpfung zu den in der ursprünglichen Datei abgespeicherten Daten.

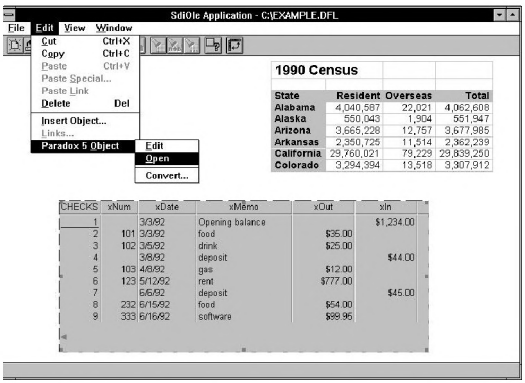
Abbildung 19.7: Das Dialogfenster Objekt einfügen vor dem Einfügen eines verknüpften Objekts



Der Text im Ergebnisfenster unten im Dialogfenster erklärt, was passiert, wenn der Benutzer OK anklickt. Das Resultat wird in Abbildung 19.8 gezeigt. Das Dokument EXAMPLE.DFL enthält nun zwei OLE Objekte - das eingebettete Quattro Pro Arbeitsblatt und die verbundene Paradox Tabelle.

Keines der beiden Objekte ist aktiv. Das Arbeitsblatt ist inaktiv, und die Datenbanktabelle ist ausgewählt. Da die Datenbanktabelle verbunden ist, zeichnet ObjectWindows ein Auswahlrechteck mit einer unterbrochenen Linie. Vegleichen Sie bitte das Auswahlrechteck in Abbildung 19.8 mit der für ein eingebettetes Objekt in Abbildung 19.6.

Abbildung 19.8: Die neue Liste der Aktionsarten für das gerade verknüpfte Objekt

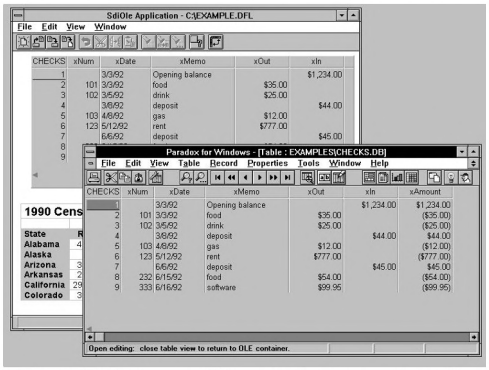


Öffnen eines Objekts zum Bearbeiten

Das Menü Bearbeiten in Abbildung 19.8 beinhaltet die Aktionsarten für die ausgewählte Paradox Tabelle. Bearbeiten und Öffnen sind die am meisten verwendeten Aktionsarten, und sowohl Quattro Pro als auch Paradox verwenden beide. Die Auswahl der Aktionsart Öffnen produziert den in Abbildung 19.9 dargestellten Bildschirm. Die gleiche Tabelle wird in zwei Fenstern sichtbar - das Container-Fenster, wo die Verknüpfung besteht, und das Server-Fenster, wo die Tabelle editiert wird. Wenn der Benutzer das Bearbeiten im Server-Fenster beendet hat, dann wählt er Datei/Schließen an und kehrt zum Container zurück. Alle Änderungen während der Editiersitzung erscheinen danach sofort im Container-Fenster.

Vergleichen wir diese Bearbeitungssitzung mit der in Abbildung 19.3 dargestellten direkten Bearbeitungssitzung. In dieser Sitzung bleibt das Container-Fenster unverändert. Das SdiOle-Fenster besitzt nur seine eigenen Befehle und seine eigene Mauspalette. Das Bearbeiten geschieht in einem eigenen Fenster, das OLE eigens für diese Sitzung eröffnet hat. Die Rückkehr zum Server zum Bearbeiten heißt *open editing* (offenes Bearbeiten). Einige Server unterstützen nur offenes Bearbeiten, nicht aber direktes Bearbeiten.

Abbildung 19.9: Ein zum Bearbeiten geöffnetes Objekt



Diese Folge von Illustrationen zeigt die üblichen Verknüpfungs- und Einbettungsoperationen. Der Benutzer verknüpft und bettet ein Objekt ein, wählt es aus, editiert es offen oder an Ort und Stelle, aktiviert es und spei-

chert das ganze Dokument vollständig mit seinen OLE-Objekten ab. Die Beispiele zeigten, wie Objekte mit dem Dialogfenster Objekte einfügen verknüpft und eingebettet werden. Es gibt dafür aber noch andere Möglichkeiten. Die Befehle Paste (Einfügen), Paste Special (Manuelles einfügen) und Paste Link (Verknüpfungen einfügen) können alle OLE Objekte mit Daten aus der Zwischenablage erzeugen. Außerdem können Objekte eingebettet oder verbunden werden indem sie mit der Maus von einer Anwendung auf eine andere gezogen werden.

Was ist ObjectComponents

Die Betriebssystemerweiterung OLE 2 von Microsoft verlangt vom Benutzer, daß in Abhängkeit von den zu erfüllenden Aufgaben der Anwendung verschiedene Oberflächen implementiert werden. Borland hat bereits in mehreren kommerziellen Anwendungen verwendete OLE-Werkzeuge entwickelt, die durch Implementierung einer kleineren Menge von weitentwickelten Oberflächen auf OLE die Arbeit des Programmiers vereinfacht. Die Werkzeuge befinden sich in einer Bilbliothek mit dem Namen BOCOLE.DLL. Die BOCOLE unterstützende Bibliothek enthält voreingestellte Implementationen für viele OLE-Standardoberflächen.

C++ Programmierer können durch neue Klassen, die zusammengefaßt ObjectComponents Framework (OCF) genannt wird, die OLE-Unterstützung in BOCOLE.DLL nutzen. Anstatt

Oberflächen in der OLE-Form zu implementieren, werden Objekte aus den ObjectComponents Klassen erzeugt und ihre Arbeitsweisen genutzt. Ihre eigenen Klassen können die von OLE gebotenen Möglichkeiten einfach durch Vererbung von den ObjectComponents Klassen erlangen. ObjectComponents übersetzt von C++ nach OLE und umgekehrt.

Abbildung 19.10 zeigt, wie die Schichten der Borland OLE-Unterstützung zusammenpassen.

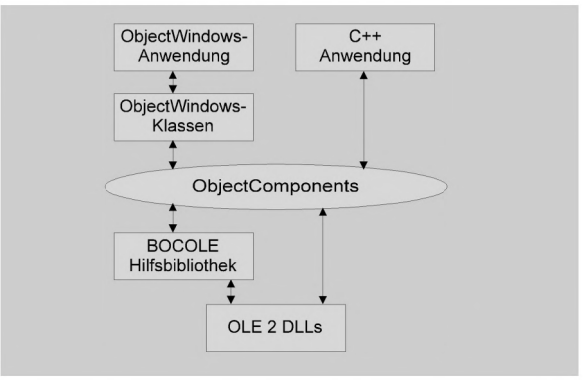


Abbildung 19.10: Wie Anwendungen mit OLE durch ObjectComponents zusammenarbeiten

Die ObjectComponents-Klassen implementieren Oberflächen in der OLE-Form, um sich mit der BOCOLE unterstützenden Bibliothek zu verständigen. Ihr Programm erreicht OLE, indem die Möglichkeiten der ObjectComponents-Klassen genutzt werden. Wenn OLE Ihnen Informationen schickt, dann sendet ObjectComponents Botschaften an Ihre Anwendung, indem es den normalen Windows Botschaftsmechanismus benutzt. Die ObjectComponents-Klassen enthalten außerdem voreingestellte Implementationen für alle OLE-Botschaften. Sie können die Voreinstellung jedoch einzeln überschreiben, um die Botschaften Ihrer Anwendung zu verändern.

ObjectComponents ist kein Teil der ObjectWindows Bibliothek. Das heißt, C++ Programme, die ObjectWindows nicht verwenden, können dennoch die durch ObjectComponents gebotenen Möglichkeiten einzubetten, zu verknüpfen und zu automatisieren voll ausnutzen. Aber ObjectWindows kann Ihre Arbeit noch mehr vereinfachen. ObjectWindows 2.5 führt neue Klassen, wie z.B. *TOleWindows* oder *TOleDocument* ein, die von ObjectComponents Klassen übernehmen, um OLE-Unterstützung in das Borland C++ Anwendungs-Framework zu bringen. Eine das Modell Dok/View nutzende ObjectWindows Anwendung braucht ObjectComponents gar nicht direkt zu verwenden. Mit einigen einfachen Änderungen an Ihrem Dok/View-Programm können Sie fast sofort verknüpfen und einbetten. Programme ohne Dok/View können dasselbe mit ein wenig mehr Arbeitsaufwand auch erreichen.

Die folgenden Kapitel erklären Schritt für Schritt, wie Sie Ihren Code verändern müssen, um Container, Server, automatisierte Objekte und Controller erzeugen zu können.

ObjectComponents unterstützte OLE 2 Merkmale

Folgende Liste faßt die den Anwendungen durch ObjectComponents gebotenen OLE 2 Möglichkeiten zusammen. Die Beschreibungen gehen davon aus, daß außerdem ObjectWindows genutzt wird. Dieselben Merkmale sind auch ohne ObjectWindows durch ObjectComponents erhältlich, aber sie erfordern explizit einige Programmierschritte, die ObjectWindows automatisch vollzieht.

- **Verknüpfen und Einbetten:** Um Daten aus einer Anwendung in einem Dokument einzubetten, bietet ObjectComponents Klassen an, damit die Daten aus dem Objekt dargestellt und Bilder der Daten auf dem Bildschirm aufgebaut werden können. Die Daten müssen von Ihrer graphischen Darstellung trennbar sein, da sie bei manchen OLE-Transaktion durch verschiedene Programme behandelt werden. Wenn der Container den Server danach fragt, ein Objekt einzubetten, dann muß der Server Daten und ein Bild der Daten liefern. Der Server kann außerdem gefragt werden, ein Objekt zu bearbeiten, obwohl es bereits eingebettet wurde, und das Objekt in oder aus dem Container-Dokument zu schrei-

ben oder zu lesen. Die Klassen von ObjectComponents bearbeiten beide Seiten dieser Aufgaben für Sie.

- **Operationen der Zwischenablage:** Die voreingestellten Ereignisverwalter für ObjectComponents-Botschaften übernehmen Schneiden und Einfügen für Sie. Wenn Sie Ihren Menüs Standardbefehle, wie z.B. Objekt einfügen oder Verknüpfung einfügen, hinzufügen, dann implementiert ObjectComponents diese für Sie.
- **Ziehen und Ablegen:** Die voreingestellten Ereignisverwalter für ObjectComponents-Botschaften helfen Ihnen auch hier. Wenn der Benutzer ein OLE-Objekt auf Ihr Container-Fenster fallen läßt, dann fügt ObjectComponents es in Ihr Dokument ein. Wenn der Benutzer das eingebettete Dokument anklickt, dann aktiviert ObjectComponents es. Wenn der Benutzer das Objekt zieht, dann bewegt ObjectComponents es.
- **Die Standard-OLE 2-Benutzeroberfläche:** OLE definiert Standardmerkmale der Benutzeroberfläche und fragt OLE-Programierer, ob sie diesen entsprechen. In ObjectComponents wurden Dialogfenster für Befehle wie Objekt einfügen, Manuell einfügen und Verknüpfung einfügen eingebaut. Außerdem gibt es ein Popup-Menü, das erscheint, sobald der Benutzer das eingebettete Objekt mit der rechten Maustaste anklickt, und einen Punkt im Menü Bearbeiten des Containers, das die für das aktive Objekt nutzbaren Aktionsarten (Server-Befehle) anzeigt. Mit ObjectComponents kann man sogar das Container-Fenster verändern, wenn der Server die Mauspalette, Statusleiste und die Menüs für das direkte Bearbeiten des Containers übernimmt.
- **Zusammengesetzte Dateien:** Eine Klasse von ObjectComponents (*TOcStorage*) umfaßt Datei-Eingabe und -Ausgabe als zusammengesetzte Dateien. Wenn Sie eine ObjectWindows-Dok/View-Anwendung in einen ObjectComponents-Container konvertieren, dann schreibt sich das Dokument automatisch in eine zusammengesetzte Datei. Dabei werden die benötigten Speicher und Unterspeicher innerhalb der Datei hergestellt. (Anweisungen für die Konvertierung finden Sie in Kapitel 20.)
- **EXE- und DLL-Server:** ObjectComponents erlaubt es Ihnen, Ihren OLE-Server als alleinstehendes ausführbares Programm oder als einen In-Process DLL-Server zu konstruieren. DLL-Server antworten Clients schneller, da sie kein separater Prozess sind. OLE muß keine Befehlsreihenfolge verwalten oder Parameter anordnen, um zwischen einem DLL-Server und seinen Clients zu kommunizieren. Weitere Informationen stehen im Abschnitt »Erzeugen eines DLL-Servers«.
- **Automatisierung:** ObjectComponents erlaubt, daß C++ Klassen ohne strukturelle Veränderungen in den Klassen selbst automatisiert werden. Dies wird durch verschachtelte Klassen erreicht, die direkten Zugriff auf die existierenden Klassenelemente haben. Diese verschachtelten Klassen benutzen kleine Befehlsobjekte, die die Elemente durch

Standardmechanismen von C++ erreichen, ohne die beschränkten und unbeweglichen Stack-Manipulationen zu benutzen. Die Befehlsobjekte unterstützen Hooks, um die Automatisierungsbefehle aufzulösen, zu speichern und zu filtern. Ein Programm kann durch Nutzung des Standardcodes von C++ sogar an sich selbst Automatisierungsbefehle senden. Kapitel 22 beschreibt die Programmierung der Automatisierung.

- **Typbibliotheken:** Eine Typbibliothek beschreibt für OLE alle nutzbaren Befehle, um eine automatisierte Anwendung kontrollieren zu können. Sobald Sie einen Automatisierungs-Server erzeugt haben, können Sie ObjectComponents bitten, die Typbibliothek für Sie herzustellen und zu registrieren.
- **Registrierung:** OLE verlangt von den Anwendungen, sich durch die OLE-Bibliotheken zu registrieren, indem Ihnen eine eindeutige Identifikationszeichenkette zugeordnet wird. Für Server muß diese Zeichenkette und eine Reihe anderer Informationen als Teil des Installationprozesses des Programms in der Registrierungsdatenbank des Systems abgelegt werden. Mit ObjectComponents müssen Sie ganz einfach alle Informationen an einer bestimmten Stelle aufzählen. Dabei werden einfache neue Makros verwendet. Jedesmal wenn Ihr Server anfängt zu arbeiten, bestätigt ObjectComponents, daß die Datenbank den aktuellen Server Inhalten entspricht. Sollte es nötig sein, so speichert oder aktualisiert ObjectComponents die Registrierungseinträge automatisch. Mehr zu Registrierungen finden Sie im Abschnitt »Registrieren eines Containers«.
- **Lokalisierung:** OLE Server müssen die Sprache Ihres Client-Programms sprechen. Wenn ein Automatisierungs-Server in mehreren Ländern verkauft wird, dann muß er Befehle in verschiedenen Sprachen verstehen. Ein Verknüpfungs- und Einbettungs-Server registriert Zeichenketten, die dem Benutzer seine Objekte beschreiben. Diese Zeichenketten sollten auch in verschiedenen Sprachen erhältlich sein, um die Anforderungen der jeweiligen Benutzer zu erfüllen. Wenn Sie Übersetzungen für Ihre Zeichenketten zur Verfügung stellen, dann benutzt ObjectComponents die richtigen Zeichenketten zum richtigen Zeitpunkt. Fügen Sie Ihre Übersetzungen den Programmquellen hinzu, und markieren Sie die ursprünglichen Zeichenketten als lokalisiert, wenn Sie diese registrieren. Zur Laufzeit holt ObjectComponents schnell und effizient die Übersetzungen zurück, damit eine Übereinstimmung mit der von OLE genutzten Sprache hergestellt wird. Mehr über Lokalisierung finden Sie im Abschnitt »Registrierung lokalisierter Einträge«.

Verwenden von ObjectComponents

Dieser Abschnitt enthält Informationen, die Ihnen helfen sollen, ObjectComponents zu verwenden. Er gibt einen Überblick über die Klassen in Botschaften in ObjectComponents, sowie den neuen Klassen in ObjectWindows, die Ihnen helfen, ObjectComponents zu nutzen. Hier wird auch erklärt, wie ObjectComponents die C++ Ausnahmenverwaltung verwendet, wie eine ObjectComponents-Anwendung erzeugt wird und welche Dateien zu Ihrer Anwendung dazu gehören.

Überblick über Klassen und Botschaften

Die folgenden Tabellen führen die am häufigsten gebrauchten ObjectComponents-Klassen und -Botschaften ein. In den nächsten Kapiteln wird ihr Gebrauch im Detail beschrieben.

Verknüpfende und einbettende Klassen

Die in Tabelle 19.3 aufgezählten Klassen unterstützen Verknüpfen und Einbetten. Sollte Ihr Programm jedoch ObjectWindows verwenden, dann brauchen Sie die meisten nicht direkt zu nutzen.

Tabelle 19.3: Einige verknüpfende und einbettende ObjectComponents Klassen

Klasse	Beschreibung
TOcApp	Verknüpft Container und Server mit OLE. Sie implementiert COM-Oberflächen für die Anwendung.
TOcDocument	Stellt ein zusammengesetztes Dokument dar. Sie hält *Teile* (eingebettete Objekte) zusammen.
TOcModule	Eine Klasse, die ein Anwendungsobjekt in einem Verknüpfungs- und Einbettungsprogramm herleitet. Sie koordiniert einige mit Registrierungs- und Hauptspeicherverwaltung verwandte, grundlegende Aufgaben.
POcPart	Stellt ein eingebettes oder verbundes Objekt in einem Dokument dar.
TOcRegistrar	Speichert Anwendungsinformationen in der Systemdatenbank ab und sagt OLE, wann die Anwendung startet oder anhält. Sie erzeugt außerdem das Objekt *TOcApp* und anwortet, wenn OLE von einem Server möchte, daß er etwas erstellt.
TOcRemView	Stellt einen Fernausschnitt für ein Server-Dokument dar. Der Server erzeugt einen Fernausschnitt für jedes an einen Container weitergegebenes Objekt. Der Fernausschnitt wird im Container-Fenster gezeichnet.
TOcView	Ist für die Darstellung eines Teils verantwortlich. Ein Container benötigt eine Darstellung für jedes eingebettete Teil.

Obwohl ObjectComponents Klassen für Dokumente und Ausschnitte verwendet, ist es für die Anwendung nicht erforderlich, das ObjectWindows-Modell Dok/View zu verwenden. Sollten Sie das Modell Dok/View verwenden, dann vereinfachen die neuen Klassen *TOleView* und *TOleDocument* die OLE-Programmierung. ObjectWindows ist jedoch keine Voraussetzung. Jedes C++ Programm kann ObjectComponents-Framework verwenden.

Verknüpfungobjekte

Einige Klassen von ObjectComponents implementieren COM-Oberflächen. (COM ist eine Abkürzung für Common Object Model. COM ist der ein OLE-Objekt definierende Standard.) Die meisten der unterstützten Oberflächen sind keine Standardoberflächen von OLE. Sie wurden speziell angefertigt und kommunizieren mit OLE mittels der BOCOLE-Hilfsbibliothek. Aber wie jedes andere COM Objekt implementieren sie *IUnknown* (durch Herleitung aus *TUnknown*).

Die Klassen, die verknüpfende und einbettende COM-Objekte definieren, sind *TOcApp, TOcView, TOcRemView* und *TOcPart*. Sie sind besondere Klassen, da sie Ihre Anwendung mit OLE verknüpfen. Sie heißen *Verknüpfungsobjekte* (*Connector Objects*). Eine ObjectComponents-Anwendung muß Verknüpfungsobjekte erzeugen, um mit anderen OLE-Anwendungen zusammenarbeiten zu können.

Da Verknüpfungsobjekte COM-Objekte sind, haben sie eine Besonderheit: Sie sind schreibgeschützt, sie können also mit Löschen nicht entfernt werden. Leser, die mit OLE vertraut sind, werden erkennen, daß die Verknüpfungsobjekte interne Referenzzähler besitzen, um die Anzahl der sie benutzenden Clients festzuhalten. Sie sind oft nicht der einzige Benutzer Ihrer eigenen Verknüpfer. Wenn z.B. ein Server eine *TOcRemView* erzeugt, um ein Objekt in einem Container-Fenster zu zeichnen, dann wird der Container ein Client desselben Objekts. Der Server darf das Ausschnittsobjekt nicht zerstören, bis der Container mit der Benutzung fertig ist. Ansonsten könnte OLE versuchen, Funktionen unter Adressen anzurufen, die gar nicht mehr existieren.

Das Common Object-Modell schreibt vor, daß ein Objekt einen internen Referenzzähler beinhalten muß. Wenn ein Client einen Zeiger zu einer der Objektoberflächen erhält, dann erhöht sich die Referenzzahl. Solange die Zahl größer als Null ist, darf das Objekt nicht gelöscht werden. Wenn die Zahl Null erreicht, dann zerstört sich das Objekt selbst.

ObjectComponents erspart Ihnen die Details der Referenzzählung. Sie müssen nie die Referenzzahl erhöhen oder erniedrigen. COM-Objekte besitzen aber Referenzzähler. Sie können sie nicht löschen, da der Befehl **Löschen** nicht mit dem Referenzzähler zusammenhängt. Benutzen Sie statt dessen die *ReleaseObject*-Methode des Verknüpfers.

Automatisierungsklassen

Die Tabelle 19.4 beschreibt einige in Automatisierungsprogrammen vorkommende Klassen.

Tabelle 19.4: Einige Automatisierungsklassen von ObjectComponents

Klasse	Beschreibung
TAutoBase	Vereinfacht Bereinigungsaufgaben, wenn ein automatisiertes Objekt gelöscht wurde. Wenn Sie diese Hilfe in Anspruch nehmen wollen, machen Sie sie Klasse zur Grundlage für Ihre Automatisierungsklassen.

TAutoProxy	Basisklasse für die Platzhalterobjekte eines Automatisierungs-Controllers. Controller erzeugen C++ Platzhalterobjekte, um die OLE-Objekte, die sie verändern möchten, darzustellen. Die Platzhalterobjekte werden mit OLE verbunden, sobald sie von *TAutoProxy* hergeleitet werden.
TOleAllocator	Lädt die BOCOLE unterstützende Bibliothek und schickt, falls erwünscht, einen speziellen Speicherplatzreservierer an OLE, um jeden Speicher zu verwalten, den das System für das Programm reserviert.
TRegistrar	Zeichnet Anwendungsinformationen in der Systemdatenbank auf und sagt OLE, wann die Anwendung startet und anhält.

Es gibt mehr Automatisierungsklassen als in der Tabelle erwähnt werden, aber viele von ihnen liegen innerhalb der ObjectComponents-Implementation. Die meiste Arbeit bei der Automatisierung einer existierenden Anwendung wird mit Makros erledigt. Eine Klasse zu automatisieren heißt, zwei Tabellen von Makros zu schreiben, eine in der Klassendeklaration und eine in der Klassen-Implementation. Die Makros beschreiben die zur Darlegung ausgewählten Methoden. Innerhalb der übergeordneten Klasse erzeugen sie eingebaute Klassen, eine für jeden Befehl. ObjectComponents weiß, wie es eine eingebaute Klasse dazu bringt, die dargelegte Methode auszuführen, und die eingebaute Klasse ruft die Elemente der übergeordneten Klasse direkt auf.

Die zur Implementation von COM-Oberflächen für ein Automatisierungsprogramm von ObjectComponents erzeugten Verknüpfungsobjekte werden als intern betrachtet. ObjectComponents stellt diese für Sie her, wenn sie gebraucht werden.

ObjectComponents Botschaften

Wenn ObjectComponents einer Anwendung Mitteilungen über von OLE kommende Signale und Ereignisse machen möchte, dann sendet es Botschaften über die normalen Windows Warteschlangen. Die gesendete Botschaft heißt WM_OCEVENT. Der Wert in *wParam* der Botschaft bezeichnet ein bestimmetes Ereignis. Nur Verknüpfen und Einbetten unterstützende Anwendungen können WM_OCEVENT Botschaften empfangen. (Diese werden durch die *TOcApp, TOcView* und *TOcRemView* Objekte der Anwendung ausgesendet. Automatisierungsanwendungen haben keine Verwendung für die Objekte.)

Einfache ObjectWindows-Anwendungen müssen keine Ereignisse verarbeiten, da die neuen OLE-Klassen voreingestellte Verwalter besitzen, die verständliche Antworten geben. Um das voreingestellte Verhalten zu verändern, fügen Sie Ihren ObjectWindows-Programmen Ereignisverwalter hinzu oder schauen Sie nach, ob WM_OCEVENT in Ihrer Windows Prozedur existiert. Weitere Informationen über die Bearbeitung von Ereignissen in einem ObjectWindows-Programm finden Sie im Kapitel über »Bearbeitung von Ereignissen«.

Die Ereignisse werden in zwei Gruppen aufgeteilt. Diejenigen, die eine Anwendung im Ganzen betreffen, stehen in Tabelle 19.5. Ereignisse, die eine Antwort aus einem bestimmten Dokument brauchen, sind an das Sichtfenster adressiert. Sie finden sie in Tabelle 19.6.

Tabelle 19.5: Anwendungsbotschaften für TOcApp-Clients

wParam-Wert	Empfänger	Beschreibung
OC_APPDIALOHELP	Container	Befiehlt dem Container Hilfe für eine der Standarddialogfenster anzuzeigen, wenn der Benutzer den Hilfeschalter angeklickt hat.
OC_APPBORDERREQ	Container	Fragt den Container, ob er dem Server Rahmenplatz in seinem Rahmenfenster geben kann.
OC_APPBORDERSPACESET	Container	Befiehlt dem Container, dem Server Rahmenplatz in seinem Rahmenfenster zu geben.
OC_APPCREATECOMP	Server (und Container als Verknüpfungsquelle)	Befiehlt der Anwendung eine neue Komponente für die Einbettung in einer anderen Anwendung zu erzeugen.
OC_APPFRAMERECT	Container	Fragt die Koordinaten für das innere Rechteck des Hauptfensters des Programms ab.
OC_APPINSMENUS	Container	Befiehlt dem Container, seine Menüs in die des Servers einzufügen.
OC_APPMENUS	Container	Befiehlt dem Container, die eingefügten Menüs in die Menüleiste einzutragen.
OC_APPPROCESSMSG	Container	Befiehlt dem Container, Beschleuniger und andere Botschaften aus der Warteschlange des Servers zu verarbeiten.
OC_APPRESTOREUI	Container	Sagt dem Container, sein normales Menü und seine normalen Grenzen wiederherzustellen, da die direkte Bearbeitung beendet wurde.
OCAPPSHUTDOWN	Server	Sagt dem Server, wann das zuletzt eingebettete Objekt schließt. Wenn der Server nichts anderes zu tun hat, kann er abschließen.
OC_APPSTATUSTEXT	Container	Schickt Text für die Statusleiste vom Server an den Container.

Ein *Ausschnitt (View)* eines Objekts ist das Abbild auf dem Bildschirm. Wenn ein OLE-Server ein Objekt an einen Container übergibt, enthält das Objekt Daten. Der Server liefert außerdem einen Ausschnitt der Daten, damit OLE das Objekt auf dem Bildschirm zeichnen kann. Manchmal bezieht sich das Wort *Ausschnitt (View)* auf das Fenster, in dem der Container ein zusammengesetztes Dokument mit all seinen eingebettetem Teilen zeichnet. Jedes Objekt besitzt seine eigenen kleinen Ausschnitte und der Container hat einen einzelnen großen Ausschnitt des gesamten Dokuments mit allen seinen eingebetteten Objekten.

Tabelle 19.6: Botschaften der Ausschnitte für TOcView und TOcRemView Clients

wParam-Wert	Empfänger	Beschreibung
OC_VIEWATTACHWINDOW	Server	Befiehlt dem Server-Fenster, sich an sein eigenes Rahmenfenster oder das Container-Fenster anzuhängen.
OC_VIEWBORDERSPACEREQ	Container	Fragt, ob der Server innerhalb des Ausschnitts eines eingebetteten Objekts Platz für eine Mauspalette haben kann.
OC_VIEWBORDERSPACESET	Container	Befiehlt dem Container, innerhalb des Ausschnitts eines eingebetteten Objekts die Mauspalette zu plazieren.
OC_VIEWCLIPDATA	Server	Befiehlt dem Server, Daten der Zwischenablage in einem bestimmten Format bereit zu halten.
OC_VIEWDRAG	Server	Befiehlt dem Server, visuelles Feedback bereitzuhalten, während der Benutzer sein eingebettetes Objekt zieht.
OC_VIEWDROP	Container	Fragt den Container, ob ein bestimmtes Objekt an einem Punkt in seinem Fenster fallengelassen werden kann.
OC_VIEWGETPALETTE	Server	Fragt den Server nach der zum Zeichnen eines Objekts benutzten Farbpalette.
OC_VIEWGETSCALE	Container	Fragt den Container nach Skalierungsinformationen.
OC_VIEWGETSITERECT	Container	Fragt den Container nach dem Lagerechteck, das einen Teil besetzt.
OC_VIEWINSMENUS	Server	Befiehlt dem Server, seine Menüs in eine gemischte Menüleiste einzutragen.
OC_VIEWLOADPART	Server	Befiehlt dem Server, ein in einer Datei des Containers abgespeichertes eingebettetes Objekt zu laden.
OC_VIEWOPENDOC	Server	Fragt den Server nach den Ausmassen seines offenen Dokuments.
OC_VIEWPAINT	Server	Befiehlt dem Server, ein Objekt an einer bestimmten Stelle in Zusammenhang mit einem gegebenen Gerätekontext zu zeichnen oder erneut zu zeichnen.
OC_VIEWPARTINVALID	Container	Teilt dem Container mit, daß eines seiner eingebetteten Objekte erneut gezeichnet werden muß.
OC_VIEWPARTSIZE	Server	Fragt den Server nach der Größe seines Ausschnitts in Pixel.
OC_VIEWSAVEPART	Server	Befiehlt dem Server, Daten für ein Objekt in die Container-Datei zu schreiben.
OC_VIEWSCROLL	Container	Befiehlt dem Container, sein Ausschnittssfenster zu durchlaufen.
OC_VIEWSETSCALE	Server	Befiehlt dem Server, eine Skalierung zu verwalten.
OC_VIEWSETSITERECT	Container	Befiehlt dem Container, das Ortsrechteck zu setzen.
OC_VIEWSHOWTOOLS	Server	Befiehlt dem Server, seine Mauspalette im Container-Fenster anzuzeigen.
OC_VIEWTITLE	Container	Fragt den Container nach dem Titel in seinem Rahmenfenster.

Die meisten Ereignisse in den Tabellen 19.5 und 19.6 werden nur an einen Server oder einen Container gesendet. Eine einzelne Anwendung erhält beide Arten von Botschaften, wenn sie sowohl Server- als auch Container-Fähigkeiten unterstützt.

Botschaften und Fenster

Da die Ausschnitts- und Teilobjekte erwarten, daß sie Botschaften an ein bestimmtes Dokument schicken müssen, wird von jeder ObjectComponents-Anwendung erwartet, daß sie ein neues Fenster für jedes offene Dokument erzeugt. Dokumentenfenster sollten keine Rahmenfenster sein; sie sollten Client-Fenster sein, die den Client-Platz eines übergeordneten Rahmenfensters genau ausfüllen. In einer SDI-Anwendung ist das übergeordnete Fenster das Hauptrahmenfenster der Anwendung. In einer MDI-Anwendung ist die Überordnung ein MDI-Unterordnungsrahmen. ObjectWindows-Programme sollten *TOleFrame* für die Client-Fenster verwenden. Viele ObjectWindows-Anwendungen, einschließlich all derer die das Dok/View-Modell verwenden, besitzen bereits Client-Fenster. Hilfe für die Implementierung von Client-Fenstern mit ObjectWindows finden Sie in »3. Erstellen des Client-Fensters«. Um Client-Fenster in einem C++ Programm zu implementieren, siehe »Erzeugen eines Ausschnittsfensters«.

Neue ObjectWindows-OLE-Klassen

Eine andere Reihe neuer Klassen integriert ObjectWindows mit ObjectComponents. Die neuen ObjectWindows-Klassen verwenden die ObjectComponents-Klassen intern, um für Sie mit OLE zu verknüpfen. Abhängig von der Komplexität Ihrer ObjectWindows-Anwendung müssen Sie nicht unbedingt direkt mit ObjectComponents zusammenarbeiten. Tabelle 19.7 faßt die wichtigsten neuen ObjectWindows-Klassen kurz zusammen.

Tabelle 19.7: Neue Klassen in ObjectWindows für die OLE-Unterstützung

Neue Klassen	Basisklassen	Beschreibung
TOleFrame	TDecoratedFrame	Liefert OLE-Unterstützung der Benutzeroberfläche für das Hauptfenster einer SDI-Anwendung.
TOleMDIFrame	TMDIFrame und TOleFrame	Liefert OLE-Unterstützung der Benutzeroberfläche für das Hauptfenster einer MDI-Anwendung.
TOleWindow	TWindow	Wird als Client eines Rahmenfensters verwendet und liefert Unterstützung für die Einbettung von Objekten in einem zusammengesetzten Dokument.
TStorageDocument	TDocument	Bietet die Fähigkeit, mit der zusammengesetzten Dateistruktur von OLE zu arbeiten. Dies ist die natürliche Auswahl für zusammengesetzte Dokumente mit eingebetteten Objekten.
TOleDocument	TStorageDocument	Implementiert die Dokumentenhälfte, von eines OLE ermöglichten Dok/View-Paares.

TOleView	TOleWindow, TView	Implementiert die Ausschnittshälfte eines Dok/View-Paares. (Informationen über Dok/View-Paare stehen in Kapitel 11.)
TOleFactory, TOleDocViewFactory, TAutoFactory,...	(nicht dokumentiert)	Implementiert die Funktionen, die von OLE aufgerufen werden, sobald eine Anwendung ein Objekt erzeugen soll. (Es gibt sechs Factory-Klassen-Template.)

Die ObjectWindows-OLE-Klassen erzeugen ObjectComponents-Objekte, wie sie benötigt werden. Wenn z.B. ein Container oder ein Server ein zusammengesetztes Dokument erzeugt, dann wird auch ein Objekt *TOcView* (oder *TOcRemView*) erzeugt, um die Oberflächen zu implementieren, die ein Dokument an OLE binden. *TOleView::CreateOcView* erledigt das für Sie. Wenn das neue Objekt *TOcView* Ereignisbotschaften an das Ausschnittfenster schickt, dann verarbeitet *TOleView* diese mit Verwaltern wie *EvOcViewSavePart* und *EvOcViewInsMenus* für Sie. Die voreingestellten Verwalter bearbeiten einen Großteil der Benutzeroberfläche für Sie.

Exception Handling in ObjectComponents

ObjectWindows 2.5 verändert die Hierarchie der Exception-Klassen. *TXBase* ist die neue Basisklasse für alle Exception-Klassen. *TXOwl* wird daraus hergeleitet, wie die in Tabelle 19.8 zusammengefaßten Exception-Klassen.

Tabelle 19.8: ObjectComponents Exception-Klassen

Klasse	Zweck
TXAuto	Während der Automatisierung auftretende Exception
TObjComp	Während ObjectComponents verknüpfender und einbettender Operationen auftretende Exception
TXOle	Während der Ausführung von OLE API Befehlen auftretende Exception
TXRegistry	Während der Benutzung der Systemdatenbank auftretende Exception

Da die Ausnahmeklassen alle aus *TXBase* hergeleitet werden, wählt die allgemeine Aussage **Catch** oft ein *TXBase&* als Parameter. Die allgemeine Aussage **Catch** im folgenden Beispiel erhält jede in ObjectWindows oder ObjectComponents gültige Ausnahme:

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
   try {
      Registrar = new TOcRegistrar(AppReg, TOleFactory<yApp>(),
                                   TApplication::GetCmdLine());
      return Registrar->Run();
   }
   catch (TXBase& x) {
```

```
      ::MessageBox(0, x.why().c_str(), "Exception", MB_OK);
   }
   return -1;
}
```

TXOLe und OLE-Fehlercodes

Die meisten OLE API-Funktionen schicken einen Wert vom Typ HRESULT zurück (oder den fast identischen SCODE). Dieser Wert gibt an, ob der Anruf erfolgreich war, und kann außerdem andere Statusinformationen enthalten. Wenn eine öffentliche Elementfunktion einer ObjectComponents Klasse in einem Anruf an die OLE-Oberfläche resultiert und dieser Anruf scheitert, dann verwandelt ObjectComponents das OLE-Rückgabeergebnis in ein C++ Ausnahmeobjekt vom Typ *TXOle*. Das erlaubt Ihnen, OLE Fehlercodes über die Standardkonstruktionen **try** und **catch** zu bearbeiten.

Die Klasse TXOle definiert die Variable Stat, die den von einem gescheiterten OLE API-Aufruf zurückgegebenen Wert enthält. Damit hat ein TXOle& als Parameter verwendender *Ausdruck* **Catch** zugriff auf den OLE-Fehlercode. Das folgende Programm zeigt ein Beispiel einer Routine, wobei der Fehlerwert einfach an den Aufrufer zurückgegeben wird. Das ist bei einer Funktion, die von einer Anwendung ohne C++ Ausnahmenbehandlung aufgerufen worden ist, sehr nützlich.

```
HRESULT
TMyAppDescriptor::CheckTypeLib(TLangId lang, const char far* file)
{
   HRESULT stat = HR_NOERROR;

   // Create OCF classes and invoke OCF methods to perform operation
   try {
      TOleCreateList typeList(new TTypeLibrary(*this, lang), file);
      f
   }

   catch(TXOle& x) {                        // Catch OLE exception
       stat = ResultFromScode(x.Stat);      // Create HRESULT from SCODE
   }
   return stat;                             // Return OLE error code
}
```

Das vorhergehende Beispiel verwendet das Makro Res*ultFromScode*, um ein HRESULT aus einem SCODE zu erzeugen. Die OLE-Headre definieren verschiedene andere Makros, die Ihnen erlauben, die verschiedenen Komponenten des von dem OLE API-Aufruf zurückgeschickten Wertes aufzuspalten, zusammenzusetzen und zu konvertieren. Weitere Informationen finden Sie unter der Überschrift »Fehlerverwaltende Funktionen und Makros«.

Wenn ObjectComponents eine Ausnahme *TXOle* intern findet, dann erscheint ein Dialogfenster, das den OLE-Rückgabewert enthält. OLE dokumentiert die Codes nur in der Headerdatei scode.h. Um die existierenden Informationen zugängiger zu machen, extrahiert die Datei DOCS/OLEERR.TXT Informationen aus dem Header und stellt die Codes in numerischer Reihenfolge dar.

Erzeugen einer ObjectComponents-Anwendung

Alle ObjectComponents-Anwendungen erfordern Exception-Handling und RTTI. Setzen Sie also keine Compiler-Optionen fest, die diese Merkmale außer Kraft setzen.

Verknüpfen und Einbetten von Anwendungen müssen das Speichermodell Large nutzen. Automatisierungsanwendungen können auch das Speichermodell Medium verwenden (sie laufen hier schneller).

Die IDE setzt automatisch die richtigen Compiler und Linker-Optionen für Sie fest, wenn Sie OCF im TargetExpert auswählen.

Um ein beliebiges ObjectComponents-Programm aus der Kommandozeile herzustellen, erzeugen Sie eine kurze Make-Datei, die OWLOCFMK.GEN aus dem Unterverzeichnis EXAMPLES enthält. Sollten Sie in Ihrer Anwendung ObjectWindows nicht verwenden, wird statt dessen OCFMAKE.GEN benutzt. Hier ist als Beispiel die Make-Datei, die das Programm AutoCalc herstellt:

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj autocalc.obj
HLP = MYPROGRAM
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES und OBJRES enthalten den Namen der zu erzeugenden Datei und die Namen der Objektdateien, aus denen sie erzeugt wird. HLP ist optional. Verwenden Sie es, wenn Ihr Projekt eine Online-Hilfedatei enthält. Schließlich sollte Ihre Make-Datei OWLOCFMK.GEN oder OFCMAKE.GEN enthalten. Nennen Sie Ihre Datei MAKEFILE und geben Sie folgenden Befehl in die Kommandozeile ein:

```
make MODEL=l
```

MAKE wird, unter Verwendung von Instruktionen in der eingeschlossenen Datei, eine neue für Ihr Projekt maßgeschneiderte Make-Datei herstellen. Die neue Make-Datei heißt WIN16L*xx*.MAK. Die letzten beiden Ziffern des Namens erklären, ob das Make-Datei diagnostizierende oder fehlersuchende Versionen der Bibliotheken herstellen. *01* steht für eine fehlersuchende Version, *10* für eine diagnostizierende Version und *11* für beides. Derselbe Befehl führt dann auch die neue Make-Datei aus und stellt das Programm her. Wenn Sie den Befehl ändern, um das Modell als

d zu definieren, dann ist die neue Make-Datei WIN16D*xx*.MAK und es stellt das Programm als ein DLL her.

Weitere Informationen über die Nutzung von OCFMAKE.GEN und OWLOCFMK.GEN stehen in der Anleitung am Anfang von MAKEFILE.GEN im selben Unterverzeichnis.

Tabelle 19.9 zeigt die Bibliotheken, mit denen ein ObjectComponents-Programm eine Verknüpfung eingeht.

Tabelle 19.9: Bilbiotheken für die Erstellung von ObjectComponents-Programmen

Speichermodell Medium	Speichermodell Large	DLL Bilbiotheken	Beschreibung
OCFWM.LIB	OCFWL.LIB	OCFWI.LIB	ObjectComponents
OWLWM.LIB	OWLWL.LIB	OWLWI.LIB	ObjectWindows
BIDSM.LIB	BIDSL.LIB	BIDSI.LIB	Klassen Bibliotheken
OLE2W16.LIB	OLE2W16.LIB	OLE2W16.LIB	OLE System DLLs
IMPORT.LIB	IMPORT.LIB IMPORT.LIB	Windows System	DLLs
MATHWM.LIB	MATHWL.LIB	Math Unterstützung	
CWM.LIB	CWL.LIB	CRTLDLL.LIB	C Laufzeitbibliotheken

Die ObjectComponents-Bibliothek muß vor der ObjectWindows-Bibliothek eingebunden werden.

Weitergeben von Dateien mit Ihrer Anwendung

Wenn Sie Ihre Anwendung weitergeben, dann müssen Sie außerdem einige von ObjectComponents gefordete Bibliotheken mitschicken. Ihr Installationsprogramm sollte die Dateien für den Benutzer installieren, ohne dabei aktuellere Versionen, die der Benutzer bereits in Betrieb hat, zu überschreiben.

Folgende Dateien sind Bestandteile von OLE 2 und sollten mit jeder OLE-Anwendung weitergegeben werden, egal ob der Benutzer ObjectComponents verwendet oder nicht.

```
compobj.dll     ole2conv.dll      ole2disp.dll
ole2.dll        ole2nls.dll       ole2prox.dll
storage.dll     typelib.dll       stdole.tlb
ole2.reg
```

Mit jedem Programm, das ObjectComponents verwendet, sollte außerdem die Bibliothek BOCOLE.DLL weitergegeben werden.

Des weiteren sollten bei Verwendung der DLL-Version von OWL, von Bibliotheken der Container-Klassen oder der Laufzeitbibliothek auch diese weitergegeben werden.

Alle Dateien gehören alle in das Benutzerverzeichnis WINDOWS/SYSTEM. Rufen Sie RegEdit auf, um OLE2.REG in die Registrierungsdatenbank des Benutzers einzufügen. (RegEdit, der Registrierungseditor, wird mit Windows geliefert.)

Wie ObjectComponents arbeitet

Die Informationen aus diesem Kapitel sind nicht unbedingt erforderlich, um mit ObjectComponents zu arbeiten. Sie erklären was im Hintergrund passiert, wenn ObjectComponents-Verknüpfungsobjekte erzeugt werden.

Die Hauptfunktion von ObjectComponents ist es, eine Verbindung mit OLE herzustellen. ObjectComponents ist eine zwischengelagerte Schicht zwischen OLE einerseits und Ihrem C++ Programm andererseits.

Wie ObjectComponents mit OLE kommuniziert

Jede Zusammenarbeit mit OLE erfordert die Implementation der OLE-Standardschnittstellen, wie z.B. die durch das Common Object-Modell (COM) definierten *IUnknown* und *IDispatch*.

Eine *Schnittstelle* ist einfach eine Menge von verwandten Prototypen von Funktionen. Alle OLE-Objekte, die dieselbe Schnittstelle implementieren, können die vorgeschriebenen Funktionen auf ihre eigene Art einsetzen. Wichtig ist nur, daß die Schnittstellen-Funktionen immer dieselben Parameter akzeptieren und dasselbe Ergebnis liefern. Das ermöglicht es jedem OLE-Objekt, eine beliebige Standardfunktion in jedem anderen die Schnittstelle unterstützenden OLE-Objekt aufzurufen.

Jedes OLE-Objekt muß die Schnittstelle *IUnknown* implementieren. Eine der drei Funktionen in der Schnittstelle *IUnknown* heißt *QueryInterface*. Diese in alle OLE-Objekten implementierte Funktion erlaubt die Abfrage, ob das Objekt andere gewünschte Schnittstellen wie z.B. Automatisierungs-Schnittstellen oder Datenübertragung-Schnittstellen unterstützt. Dies ermöglicht es jedem OLE-Objekt während der Laufzeit zu bestimmen, was jedes andere OLE-Objekt kann.

OLE definiert eine große Anzahl von Standardschnittstellen, die ansonsten mit erheblichem Aufwand zu implementieren wären. Die BOCOLE-Hilfsbibliothek von Borland definiert einige andere speziell erstellte COM Schnittstellen, die alle zusammen eine alternative Schnittstelle auf einem höheren Abstraktionsniveau zur OLE-Programmierung liefern. Client-Objekte der Hilfsbibliothek müssen zwar immer noch wie alle COM-Objekte *IUnknown* implementieren, aber anstelle anderer OLE-Standardschnittstellen wie z.B. *IDataObject* und *IMoniker* implementieren sie von BOCOLE definierte Schnittstellen. Die Hilfsbibliothek benimmt sich wie ein Agent, der durch seine speziell erstellten Schnittstellen Befehle er-

hält und diese in Standard-OLE übersetzt. Alle Befehle für die speziell erstellten Schnittstellen werden für die Benutzung der Standardschnittstellen von OLE ausgeführt.

Die speziellen Schnittstellen in der BOCOLE Hilfsbibliothek haben Name wie z.B. *IBContainer* und *IBDocument*. Wenn Sie im ObjectComponents-Quellprogramm nachsehen, können Sie ihren Gebrauch sehen. Da die Hilfsbibliothek ein internes Hilfmittel ist und ständigen Änderungen unterworfen ist, sind ihre Schnittstellen nicht dokumentiert. Allerdings wird das gesamte Bibliothekenprogramm mit Borland C++ ausgeliefert. Sie können also dort nachsehen, wenn Sie genaue Informationen brauchen. Außerdem können Sie bei Bedarf die Hilfsbibliothek genauso wie die ObjectWindows-Bibliothek verändern oder neu erstellen.

Wie ObjectComponents mit Ihnen kommuniziert

Einige ObjectComponents-Klassen definieren COM-Objekte. Diese Objekte werden aus *TUnknown* hergeleitet, einer ObjectComponents-Basisklasse, die die Schnittstelle *IUnknown* implementiert und die Details der Aggregation (eine Art mehrere Objekte in einer einzigen funktionalen Einheit zusammenzufassen) behandelt. Sie mischen außerdem in anderen Basisklassen mit, die Schnittstellen aus der BOCOLE-Hilfsbibliothek implementieren.

Die COM-Schnittstellen implementierenden Objekte von ObjectComponents heißen *Connector Objects* (Verbinderobjekte), da sie Ihre Anwendung mit OLE verbinden. So ist z.B. *TOcPart* jenes Verbinderobjekts, das die Schnittstelle implementiert, welches ein Container für jedes in seinem Dokument vorkommenden OLE-Objekt (Teil) unterstützen muß. Um ein Objekt in Ihrem Dokument einzubetten, benutzen Sie Informationen, die ObjectComponents aus der Zwischenablage, einer Botschaft oder aus dem Dialogfenster Objekt einfügen erhält, und leiten diese an den Konstruktor *TOcPart* weiter. Dieser Konstruktor ruft unter anderem eine BOCOLE-Funktion (indirekt) auf, um ein eingebettetes OLE-Objekt zu erzeugen. *TOcPart* enthält den Zeiger zu diesem Objekt, fragt nach den Schnittstellen, und speichert die Koordinaten der Stelle, an der das Teil gezeichnet werden soll. Wenn Sie möchten, daß das Teil etwas tut, dann rufen Sie Methoden *TOcPart*, wie z.B. *Activate* und *Save* auf.

Verbindungen verknüpfen und einbetten

Eine verknüpfende und einbettende Anwendung erzeugt immer ein Objekt *TOcApp* (normalerweise wird dieses für Sie erzeugt). *TOcApp* ist ein Verbinderobjekt, das Schnittstellen implementiert, die jede verknüpfende und einbettende Anwendung braucht. Ein anderes durch alle verknüpfenden und einbettenden Anwendungen erzeugtes Verbinderobjekt ist das Ausschnittsobjekt. Entweder heißt es *TOcView* für einen Container oder *TOcRemView* für einen Server. Für jedes geöffnete Dokument erzeugen Sie ein Ausschnittsobjekt. Ein Ausschnittsobjekt ist mit dem Fenster, in dem das Dokument gezeichnet wird, assoziiert. Das einzige andere für

Verknüpfen und Einbetten benutzte Verbinderobjekt ist *TOcPart*, das von den Containern für jedes in Ihre Dokumente geschriebenen Objekte erzeugt wird.

Natürlich ist die Kommunikation durch ein Verbinderobjekt nicht nur einseitig. Wenn Sie mit einem Verbinderobjekt arbeiten, dann ruft das Objekt OLE an, aber manchmal muß OLE Sie anrufen. Wenn z.B. der Benutzer ein eingebettetes Objekt editiert, dann muß der Server dem Container mitteilen, daß das Objekt neu gezeichnet werden muß. Der Server teilt dies OLE mit, und OLE informiert den Container. Der Container setzt seinen Teil außer Kraft, und OLE signalisiert dem Server zu zeichnen. Egal ob Sie ein Container oder ein Server sind, ruft OLE Sie an.

Die Verknüpfungsobjekte können natürlich Ihre Funktionen nicht auf demselben Weg aufrufen wie Sie seine, da diese Ihr Programm nicht kennen. Wenn ein Verknüpfungsobjekt Ihnen von OLE eine Anfrage oder eine Botschaft mitteilen muß, dann sendet es eine Botschaft WM_OCEVENT an eines Ihrer Fenster. *TOcApp* schickt seine Botschaften an Ihr Rahmenfenster. Die Ausschnitts- und Teilobjekte senden Botschaften an das Client-Fenster, in dem Sie Ihr Dokument zeichnen.

Kommunikation von Ihnen zu OLE geschieht durch Funktionsaufrufe an die Verbinderobjekte, Kommunikation von OLE zu Ihnen durch Botschaften von den Verbinderobjekten an Ihre Fenster. Die Abbildung 19.11 stellt diese Verbindungen dar.

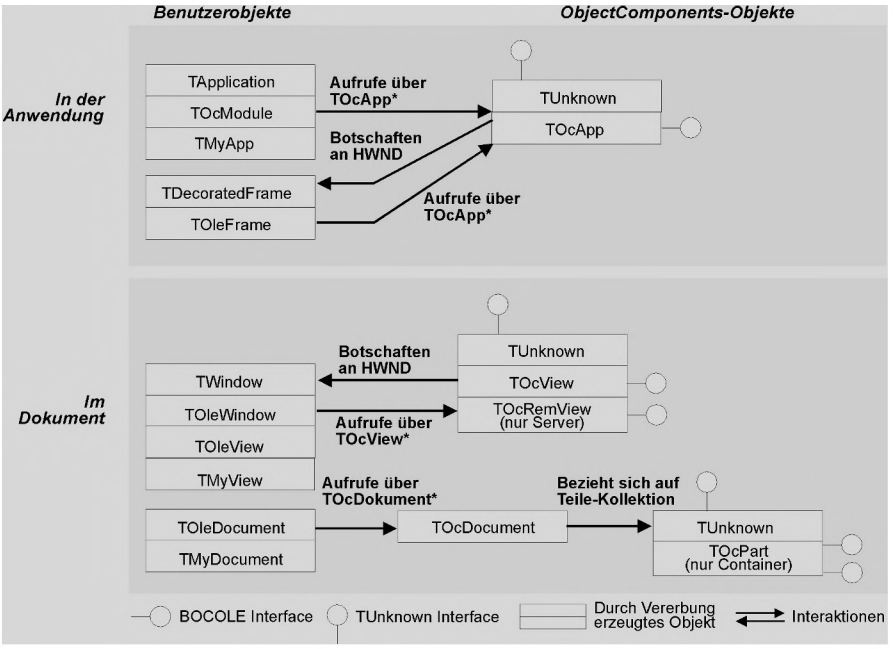


Abbildung 19.11: Wie Objekte Ihrer Anwendung mit ObjectComponents zusammenarbeiten.

Die Objekte auf der linken Seite sind Fälle der normal erzeugten ObjectWindows-Klassen: eine Anwendung, ein Rahmenfenster, ein Dokument und einen Ausschnitt. In Anwendungen ohne das Dok/View-Modell oder ohne ObjectWindows erfüllen verschiedene Klassen dieselben Funktionen. Es gibt z.B. immer ein Dokumentenfenster oder ein Rahmenfenster. Der Arbeitsfluß ist in jeder ObjectComponents-Anwendung derselbe.

Die Objekte auf der rechten Seite sind Helfer von ObjectComponents, die korrespondierende Teile Ihrer Anwendung mit OLE verbinden.

Die Anfangsleitung zwischen Ihnen und ObjectComponents wird gelegt, wenn das Registrierungsobjekt zum ersten Mal Ihre Callback-Funktion *Factory* aufruft. Das Objekt *TOcApp* ist an ein Fenster in *TOleFrame::SetupWindow* oder im Handler WM_CREATE Ihres Hauptfensters gebunden.

Automatisierungsverbindungen

Anwendungen, die Automatisierung, nicht aber Verknüpfen und Einbetten unterstützen, verwenden eine andere Menge von Objekten. Die zentrale Funktion der Automatisierungsschicht in OLE ist es, Argumente vom Controler an den Server zu übergeben, einer Operation ohne Benutzeroberfläche. Die COM Schnittstellen für die Automatisierung sind tiefer in den Implementationen von ObjectComponents eingegraben als jene für das Verknüpfen und Einbetten.

Um Automatisierung zu unterstützen, muß ObjectComponents dargelegte Befehle und Argumente erkennen, sie mit Typeninformationen belegen, Werte zum und vom Stack der Variante *VARIANT* transferieren, die OLE benutzt um Werte zu übergeben, und Ihre C++ Funktionen aufrufen, sobald ein Controller einen Befehl sendet. Sobald Sie die Tabellen gesetzt haben, die beschreiben, was Sie darlegen wollen, gibt es im Automatisierungsprozess wenig, was speziell erstellt oder überschrieben werden muß. Die Verbinderobjekte werden für die Automatisierung niemals direkt erzeugt oder verändert; ObjectComponents erledigt dies für Sie.

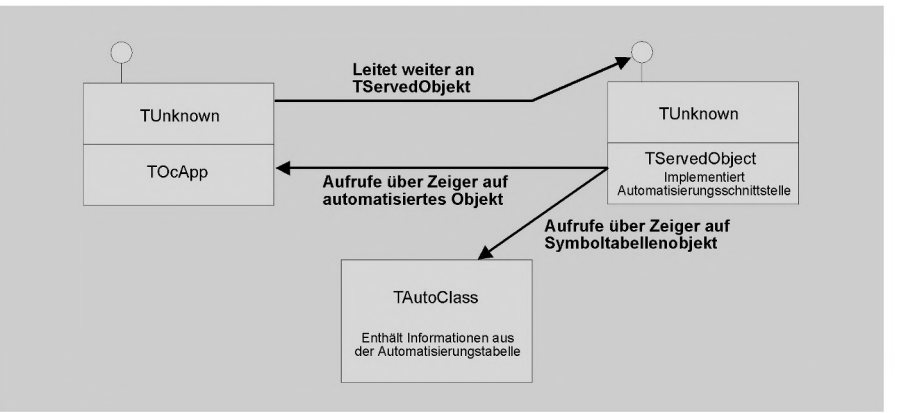
Fortgeschrittene Benutzer, die gerne Programme lesen, interessiert vielleicht, daß *TServedObject* die Klasse ist, die *IDispatch* und *ITypeInfo* implementiert, *TTypeLibrary* implementiert *ITypeLib*, und *TAutoIterator IEnumVARIANT* implementiert. Nur *TAutoIterator* wird als ein öffentlicher Teil von ObjectComponents dargelegt. Die anderen werden als Implementationen betrachtet.

Um eine Klasse zu automatisieren, fordert ObjectComponents Sie auf, mit Makros zwei beschreibende Tabellen zu erstellen. Eine Deklarationstabelle gehört zur Klassendeklaration und legt fest, welche Elemente für OLE zugänglich sind. Eine Definitionstabelle gehört zur Klassenimplementation und legt für die Benutzung durch die Controller bei Aufruf Ihrer Funktionen öffentliche Namen fest. Die Automatisierungsmakros erzeugen außerdem eingebaute Klassen innerhalb der automatisierten

Überordnung, eine für jede dargelegte Funktion oder jedes Datenelement. Diese eingebauten Klassen besitzen eine Ihre Funktionen aufrufendende Methode *Invoke*. Da die Klasse eingebaut wurde, hat sie mittels des normalen C++ Mechanismus direkten Zugriff auf Ihre Klasse.

TServedObject ist der Verbinder, der *IDispatch* Befehle von OLE erhält und diese in die entsprechenden Aufrufe *InVoke* übersetzt. *TServedObject* erhält die benötigten Informationen, um dies in einem Objekt des Typs *TAutoClass*, welches die Symbolinformationen aus den Automatisierungstabellen enthält, auszuführen. *TServedObject* erhält Zuteilungs-IDs, sucht diese in *TAutoClass* und benutzt die gefundenen Informationen, um von OLE gesandte Argumente aus dem Stack der Variante *VARIANT* zu extrahieren. Schließlich ruft es *Invoke* für das entsprechende eingebaute Objekt auf. Abbildung 19.12 stellt die Zusammenarbeit von *TServedObject* mit *TAutoClass* und Ihren automatisierten Klassen dar.

Abbildung 19.12: Wie TServedObject eine automatisierte Klasse mit OLE verbindet.



Programmierhilfen von ObjectComponents

AppExpert ist das größte Hilfsmittel von Borland C++, um Ihnen bei der Programmierung in ObjectComponents zu helfen. AppExpert erzeugt entsprechend Ihren Spezifikationen eine vollständige Basisanwendung. Es unterstützt sowohl Verknüpfen als auch Einbetten und Automatisierung vollständig. Es wird dazu verwendet, Container, Server, Automatisierungs-Server und Automatisierungs-Controller zu erzeugen. ClassExpert hilft Ihnen, das erstellte Programm zu verändern, damit es Ihren Ansprüchen entspricht.

Auch TargetExpert im IDE (Integrated Development Environment) unterstützt ObjectComponents. Klicken Sie die Option für OCF an und automatisch werden die richtigen Aufbauoptionen gesetzt.

Hilfsprogramme

Mit Borland C++ 4.5 werden einige neue Hilfsprogramme geliefert, die allgemeine OLE-Programmieraufgaben vereinfachen. Einige Problemlösungen werden in anderen Kapiteln im Detail behandelt.

AutoGen: Generiert Platzhalterklassen für einen Automatisierungs-Controller. Es durchsucht die Typenbibliothek einer automatisierten Anwendung und schreibt das Programm für Klassen, die ein Controller benutzt, um Befehlen Automatisierungsbefehle zu senden.

DllRun: Setzt einen DLL-Server in den ausführbaren Modus. Jeder mit ObjectComponents geschriebene DLL-Server kann auch als alleinstehende Anwendung ausgeführt werden, wenn er mit DllRun aufgerufen wird. Ein Testlauf im ausführbaren Modus erleichtert manchmal die DLL-Fehlersuche. Außerdem ermöglicht es ein einzelnes Programm, das Ihr Benutzer entweder als einen In-Process-Server, oder als eine unabhängige Anwendung laufen lassen kann, weiterzugeben.

GuidGen: Erzeugt global eindeutige Identifizierungen zur Benutzung in Registrierungsanwendungen. Jeder Server muß eine eindeutige Identifikation haben.

MacroGen: Generiert Automatisierungsmakros um Funktionen mit einer beliebigen Anzahl von Argumenten darzulegen. Die ObjectComponents-Header deklarieren Makroversionen für Funktionen mit bis zu vier Argumenten. MacroGen erspart Ihnen, die Makros manuell ändern zu müssen, um mehr Argumenten gerecht zu werden.

Register: Registriert oder löscht die Registrierung eines jeden ObjectComponents-EXE oder -DLL. Normalerweise registrieren sich die Anwendungen, sofern notwendig, selbst, sobald sie laufen oder auf Kommandozeilenschalter antworten. Manchmal müssen Programmentwickler aber wiederholt verschiedene Versionen einer Anwendung registrieren oder die Registrierung löschen. Registrierung ist besonders für DLLs nützlich, da keine Kommandozeilenschalter an ein DLL geschickt werden können.

WinRun: Dies ist ein Hintergrundprogramm, das es ermöglicht, Windows Programme aus der Kommandozeile eines DOS-Fensters zu starten. WinRun erlaubt auch GUI-Programmen, wie z.B. Register, aus einer Make-Datei laufen zu lassen.

Die Quellcodes für alle Hilfsprogramme außer WinRun befinden sich im Verzeichnis OCTOOLS.

Wo finde ich Informationen?

Sie finden Informationen zur Programmierung mit ObjectComponents in diesem und anderen Büchern, in der Online-Hilfe und in den Unterverzeichnissen der Beispielprogramme.

In dieser Dokumentation bezieht sich *OLE* auf OLE 2.5, außer Version 1 wird explizit angegeben.

Bücher

Die folgenden Kapitel beschreiben wie man Programme, die alle diese Funktionen erfüllen, erstellt.

Tabelle 19.10: Beschreibungen der ObjectComponents-Kapitel in diesem Buch

Titel des Kapitels	Thema
Unterstützung von OLE in Borland C++	Überblick über ObjectComponents
Erzeugen eines OLE-Containers	Wie man eine Anwendung, die OLE-Objekte in seinen Dokumenten erhält, erstellt
Erzeugen eines OLE-Servers	Wie man eine Anwendung, die OLE-Objekte für Container erzeugt, erstellt
Automatisierung einer Anwendung	Wie man eine andere Programme kontrollierende Anwendung erstellt
Erzeugen eines Automatisierungs-Controllers	Wie man eine andere Anwendungen kontrollierende Anwendung erstellt

Im *ObjectWindows Referenzhandbuch* werden alle neuen OLE-verwandten Klassen und Makros in ObjectComponents und ObjectWindows behandelt.

Das Material zu ObjectComponents in diesem Buch und im *ObjectWindows Referenzhandbuch* finden Sie auch in der Online-Hilfe für Borland C++.

Die Einführung im *ObjectWindows Programmierhandbuch* entwickelt eine ganz neue Beispielanwendung. Die weiterführenden Schritte fügen OLE-Container, -Server und -Automatisierungsfähigkeiten hinzu.

Online-Hilfe

Borland C++ enthält drei zusätzliche Hilfedateien für OLE API. Zum größten Teil macht ObjectComponents-Wissen über OLE-Schnittstellen unnötig, wenn Sie aber mehr darüber wissen wollen, wie ObjectComponents funktioniert, oder wenn Ihre Anwendung fortgeschrittenes Programmieren auf der OLE-Schnittstellenebene erfordert, dann könnten Ihnen diese Dateien nützen.

Tabelle 19.11: Online-Hilfedateien mit Informationen über ObjectComponents und OLE.

Hilfedatei	Thema
OCF.HLP	ObjectComponents-Kapitel aus dem *ObjectWindows Programmierhandbuch* und dem *ObjectWindows Referenzhandbuch*
OWL.HLP	Referenzmaterial über die neuen OLE-fähigen Klassen in ObjectWindows
OLEAUTO.HLP	Übersichten und Referenzen zu OLE 2 Automatisierung
OLE2HELP.HLP	Übersichten und allgemeine Referenzen zu OLE 2
OLE2UI.HLP	Bibliotheksreferenzen für die OLE 2 Benutzeroberfläche

Beispielprogramme

Die beste Art Programmieren zu lernen, ist, sich ausführbare Programme anzusehen. Borland C++ wird mit einer Reihe von Beispielprogrammen geliefert, die ObjectComponents zeigen. Einige dieser Programme werden in der folgenden Liste beschrieben.

EXAMPLES/OCF: ObjectComponents ohne ObjectWindows

AutoCalc: Ein Automatisierungs-Server; zeichnet einen Taschenrechner auf den Bilschirm und läßt den Controller Tasten anklicken

CallCalc: Ein Automatisierungs-Controller, um den Taschenrechner in AutoCalc zu manipulieren

CppOcf: Lernprogramm in drei Schritten für Verknüpfen und Einbetten, das mit einem einfachen C++ Programm anfängt, es erst in einen Container und danach in einen Server verwandelt

Localize: Zieht übersetzte Zeichenketten aus XLAT, um Sprachdarstellungen zu reflektieren

RegTest: Registriert, prüft die Gültigkeit der Registrierung und löscht die Registrierung einer ObjectComponents-Anwendung

EXAMPLES/OWL/TUTORIAL: Beispiele für ObjectWindows-Lernprogramme

OwlOcf: Lernprogramm in drei Schritten, das mit einem einfachen ObjectComponents-Programm anfängt, es erst in einen Container und dann in einen Server verwandelt

Step14-Step17: Enthält die letzten Schritte der Lernanwendung im *ObjectWindows Programmierhandbuch*; zeigt einen verknüpfenden und einbettenden Container oder Server, einen Automatisierungs-Server oder Controller und wie sowohl Automatisierung als auch Verknüpfen und Einbetten zur selben Zeit unterstützt werden.

EXAMPLES/OWL/OCF: ObjectComponents mit ObjectWindows

MdiOle: Eine mehrdokumentige Oberflächenanwendung mit Containerfähigkeiten

SdiOle: Eine eindokumentige Oberflächenanwendung mit Containerfähigkeiten

Tic Tac Toe: Ein verknüpfender und einbettender Server

OCTOOLS: Programme für Programmierhilfen

AutoGen: Durchsucht eine Typenbibliothek und erzeugt Platzhalterklassen für einen Automatisierungs-Controller

DllRun: Führt einen DLL-Server im ausführbaren Modus aus

GuidGen: Generiert global eindeutige Identifikationen (GUIDs)

Register: Registriert einen DLL-Server

Glossar der OLE-Begriffe

Die Definitionen in dieser Liste erklären allgemeine Begriffe der OLE Programierung. Lesen Sie es als Einführung in wichtige Programmierthemen, oder lesen Sie Erklärungen nach, während sie andere Kapitel lesen.

Die Definitionen fortgeschrittener Konzepte setzen voraus, daß Sie bereits mit OLE und seine Standardoberflächen vertraut sind. Weitere Informationen über OLE finden Sie in den drei OLE-Online-Hilfedateien.

- **Aktivieren:** Der Benutzer *aktiviert* ein verbundenes oder eingebettetes Objekt, indem er es doppelt anklickt. Normalerweise veranlaßt das Aktivieren eines Objekts seinen Server dazu, seine Menüs in die des Containers einzubauen. Der Benutzer kann dann das aktive Objekt bearbeiten, ohne das Containerfenster zu verlassen. Aktivieren ist nicht das Gleiche wie Auswählen; s. auch *Auswählen*.
- **Aggregation:** Eine Art mehrere OLE-Objekte zu kombinieren, um sie als einzelnes großes Objekt zu verwenden. Objekte werden zur Laufzeit angehäuft. Sie können Objekte aggregieren, die Sie nicht selbst hergestellt haben.

  Aggregation ist eine fortgeschrittene Programmiertechnik. Damit sie funktioniert, müssen alle kombinierten Objekte denselben Zeiger *IUnknown* als Antwort auf *QueryInterface* zurückgeben. Das Hauptobjekt in der Aggregation, normalerweise äußeres Objekt genannt, häuft sich mit einem anderen Objekt an, indem es seinen eigenen Zeiger *IUnknown* weitergibt. Das zweite Objekt erinnert sich an den äußeren Zeiger, *IUnknown* und wenn es nach seinem eigenen Zeiger gefragt wird, dann gibt es als Antwort den äußeren Zeiger *IUnknown* an. Alle Anfragen nach einer Schnittstelle von dem angehäuften Objekt werden somit an das äußere Objekt weitergeleitet. Wenn das äußere Objekt eine angefragte Schnittstelle nicht unterstützt, dann leitet es die Anfrage an das

erste Objekt in einer sogenannten Kette von Hilfsobjekten weiter. Ein Client kann mittels des *IUnknown* des äußeren Objekts alle durch die Hilfsobjekte unterstützten Schnittstellen erreichen.

- **Ausschnitt:** (View) Die graphische Darstellung von Daten. Dieser Begriff wird verwendet, um zwischen der Art, in der die Daten gezeichnet werden, und den Daten selbst, meist *Dokument* genannt, zu unterscheiden. Ein einzelnes Textverarbeitungsdokument könnte z.B. drei verschiedene Ausschnitte besitzen: einen Seitenlayoutausschnitt, einen Entwurfsausschnitt mit aufwendigen Schriften und einen Ausschnitt für die Druckvoranzeige.

  In ObjectComponents erzeugen Container Ausschnitte, um ihre zusammengesetzten Dokumente zu zeichnen. Auch Server erzeugen Ausschnitte, um die von ihnen erzeugten Objekte zu zeichnen. Beide generieren ein Objekt*TOcDocument*, um die Daten und ein Ausschnittsobjekt, entweder *TOcView* oder *TOcRemView*, das das Dokument zeichnet, zu bearbeiten.

  In ObjectWindows bezieht sich Dok/View auf eine bestimmte, durch das Framework, das Daten und seine Darstellungen auch in getrennten Klassen behandelt, unterstützte Anwendungsarchitektur.

- **Auswählen:** (Select) Der Benutzer wählt ein Objekt aus, indem er es mit der Maus einmal anklickt. Das ausgewählte Objekt wird dabei nicht aktiviert und kann nicht editiert werden. Normalerweise zeigt ein Container durch Zeichnen eines Rechtecks mit Grapples um das Objekt herum an, daß ein Objekt ausgewählt wurde. (*Grapples* sind kleine Henkel, um das Objekt zu bewegen.) Der Container könnte dem Benutzer erlauben, mehrere Objekte auf einmal auszuwählen, um sie zu bewegen oder als Gruppe zu löschen. Normalerweise kann aber nur ein Objekt zur selben Zeit aktiv sein.
- **Automatisierung:** Die Fähigkeit einer Anwendung einen Satz von Befehlen zu definieren, damit andere Anwendungen diese Befehle zurücksenden können.
- **Automatisierungs-Controller:** Eine Anwendung, die Befehle an automatisierte Objekte oder Anwendungen schickt. Ein Controller wird manchmal auch Automatisierungs-Client genannt.
- **Automatisierungs-Server:** Eine Anwendung, die einige ihrer eigenen internen Funktionsaufrufe als einen Satz von Befehlen darlegt, damit andere Programme diese benutzen können. Ein automatisiertes Objekt ist das, was der Server zur Kontrolle durch andere Programme erzeugt.
- **Automatisiertes Objekt oder Anwendung** Ein OLE-Objekt, das Befehle veröffentlicht, die ihm eine andere Anwendung geschickt hat. Ein Automatisierungs-Server erzeugt Automatisierungsobjekte. Das auto-

matisierte Objekt kann die Anwendung selbst oder etwas von der Anwendung Erzeugtes sein.

- **BOCOLE-Hilfsbibliothek:** Ein DLL von OLE-Implementation-Utility-Schnittstellen, das intern ObjectComponents aufruft. Die Hilfsbibliothek implementiert eine Reihe von speziell erstellten, durch Borland entworfenen, OLE Schnittstellen. Die Datei BOCOLE.DLL sollte mit jedem ObjectComponents-Programm weitergegeben werden. Die speziellen Schnittstellen werden als intern betrachtet und sind somit nicht dokumentiert. Der Quellcode für die BOCOLE-Hilfsbibliothek ist jedoch in Borland C++ enthalten.
- **COM-Objekt** Ein Objekt, dessen Architektur mit dem Component Object-Modell, einer Spezifikation von Microsoft, die die Basis des OLE Systems darstellt, konform geht. Dies sind kurzgefaßt die Besonderheiten von COM Objekten:
- Sie kommunizieren durch vordefinierte Schnittstellen.
- Sie unterstützen alle die Schnittstelle *IUnknown*, und IUnknown enthält die Methode *QueryInterface*, um andere optionale Schnittstellen zu holen.
- Sie enthalten einen Referenzzähler für ihre Clienten und löschen sich selbst, sobald der Zähler null wird.

  Nur COM-Objekte können mit OLE kommunizieren. Einige Klassen in ObjectComponents sind COM-Objekte (s. *Verknüpfungsobjekte*). ObjectComponents erspart Ihnen die Details der Methoden von Schnittstellen, Schnittstellen-Zeigern und Referenzzählern. Es verbindet Sie mit OLE, indem vertraute C++ und Windows Programmiermodelle wie z.B. Vererbung und Botschaften benutzt werden.

- **Container:** Eine Anwendung, die OLE erlaubt, Objekte von anderen Anwendungen mit seinen eigenen Dokumenten zu verknüpfen oder in sie einzubetten. Container heißen auch Clienten des Servers, der die Objekte weitergibt.
- **DLL-Server:** Ein Server, dessen Programm eine dynamische Verknüpfungsbibliothek (Dynamic Link Library) anstelle einer ausführbaren Datei ist. Der Vorteil des DLL-Servers ist seine Geschwindigkeit. Wenn OLE einen .EXE-Server aufruft, um ein eingebettetes Objekt zu unterstützen, dann muß es in einen getrennten Prozess hinein erzeugen und aufrufen. Ein DLL andererseits ist als sein Client ein Teil desselben System-Tasks, also laufen OLE-Aufrufe aus einem Container an einen DLL-Server sehr viel schneller. Der Nachteil, einen Server als DLL und nicht als EXE zu erstellen, liegt darin, daß er nicht als unabhängiges Programm ablaufen kann und die Fehlersuche sich als schwieriger erweist. Mit ObjectComponents können die DLL-Server jedoch einfach

im ausführbaren Modus laufen. Weitere Informationen hierzu finden Sie im Abschnitt »Herstellung eines DLL Servers«.

- **Dokument:** Dieses Wort besitzt für Programmierer zwei verschiedene Bedeutungen. Erstens ist ein Dokument eine Datenmenge, die in die Anwendung bei Aufruf des Befehls Datei/Öffnen eingelesen wird. Ein Dokument kann ein Arbeitsblatt, eine Bitmap, ein Brief oder jede beliebige andere Datenmenge, die von der Anwendung als ein Ganzes behandelt wird, sein.

  Es kann manchmal sehr hilfreich sein, zwischen den Daten in einem Dokument und dem Abbild auf dem Bilschirm zu unterscheiden. Ein Arbeitsblatt z.B. kann als Datentabelle oder als Diagramm dargestellt werden. Ein einzelnes Dokument kann auf verschiedene Arten dargestellt werden. In diesem Fall bezieht sich das Wort *Dokument* nur auf die Daten, und die möglichen Darstellungsweisen nennt man *Ausschnitt* (*View*).

  ObjectWindows Programmierer sind mit der Anwendungsarchitektur *Dok/View-Modell* vertraut, die das Programm in einen Teil für die Datenverwaltung und einen für die Darstellung aufspaltet. ObjectComponents besitzt eine Dokumentenklasse und Ausschnittsklassen, sie sind aber keine Teile des Dok/View-Modells.

  Die Dokumentenklasse verwaltet die in das Dokument eingebetteten Objekte, und die Ausschnittsklassen zeichnen das Objekt auf den Bildschirm.

- **Eingebettete Objekte:** Von einer Server-Anwendung durch OLE in einem Container-Dokument eingesetzte Daten. OLE erlaubt es dem Benutzer Objekte in einen Container einzusetzen, hinein zu ziehen oder einzufügen. Wenn der Benutzer sich während dieser Aktionen entschließt, ein *eingebettetes* Objekt zu erzeugen, dann werden alle Daten in dem Objekt in das Container-Dokuement kopiert. Wird das Dokument vom Benutzer geladen oder abgespeichert, dann werden die Daten des eingebetteten Objekts mit den ursprünglichen Daten des Container in die Datei geschrieben.

  Vergleichen Sie eingebettete mit *verbundenen* Objekten, bei denen die Daten für das OLE-Objekt in einer anderen Anwendung abgespeichert wird und der Container nur eine Referenz zur Objektdatei erhält.

- **EXE-Server:** Eine Server-Anwendung, die in eine ausführbare Datei kompiliert und mit ihr verbunden wurde. Ein Server kann auch als Bibliothek erstellt werden: s. *DLL-Server*.

- **Fernausschnitt:** (Remote View) Der Ausschnitt eines eigenen Objekts, das der Server in einem Container-Fenster zeichnet. Wenn ein ObjectComponents-Server gestartet wird, um ein Objekt zu bearbeiten, das mit einem Container-Dokument verbunden ist oder dort eingebettet

ist, dann erzeugt der Server ein Objekt *TOcRemView* und ein Objekt *TOcDocument*. Das Ausschnittsobjekt zeichnet im Container-Fenster. Das Dokumentenobjekt lädt und speichert die Objektdaten.

- **GUID:** Steht für global eindeutiger Bezeichner (Globally Unique Identifier), einem 16-Byte-Wert, der eindeutig eine bestimmte Server-Anwendung oder eine Art von einem Server erzeugten Objekt identifiziert. Da der Benutzer zusammengesetzte Dokumente von System zu System tranferieren kann, braucht OLE GUID, um verbundene oder eingebettete Objekte mit Ihrem Server zu identifizieren. Hätten zwei Server dieselbe Kennung, könnte OLE versehentlich den falschen aufrufen. Jeder Server und jeder Objekttyp muß absolut eindeutig bestimmte Kennungen haben. Hilfsmittel wie GUIDGEN erzeugen diese Kennungen. Weitere Informationen finden Sie unter dem Eintrag *clsid* im *ObjectWindows Referenzhandbuch*.
- **IDispatch-Schnittstelle:** Die OLE-Schnittstelle, das alle automatisierten Objekte implementiert. Mit den vier Arbeitsweisen der Schnittstelle *IDispatch* können Informationen über automatisierte Befehle von den automatisierten Objekten abgefragt werden, Kennungen für bestimmte Befehle gesucht werden oder jeder Befehl aufgerufen werden. Siehe auch OLEAUTO.HLP Hilfedatei.
- **Direkte Bearbeitung im Containerdokument:** Ein Objekt im Container-Fenster zu bearbeiten. Während der direkten Bearbeitung im Containerdokument läßt der Container den Server seine eigenen Menüs und Mauspaletten im Container-Fenster darstellen. Der Sinn besteht darin, daß der Benutzer jedes Objekt in einem Dokument bearbeiten kann, ohne das Dokumentenfenster zu verlassen. Die direkte Bearbeitung wird in Abbildung 19.3. dargestellt. Vergleichen Sie auch *offenes Bearbeiten*.
- **In-Process Server:** Siehe DLL-Server.
- **Schnittstelle:** Ein Satz Funktionsprototypen, die normalerweise als abstrakte Basisklassen deklariert sind. OLE-Objekte können miteinander kommunizieren, da sie Standardschnittstellen implementieren, also Sätze von Funktionen, die vom System definiert werden. Das System definiert nur die in einer Schnittstelle enthaltenen Funktionen; es implementiert die Funktionen nicht. Jedes Objekt implementiert die Funktionen für sich selbst. Die Schnittstellen werden in den OLE-System-Headers, wie z.B. compobh.hj und ole2.h definiert. Das OLE-System kommuniziert mit Anwendungen und Objekten durch Aufruf der implementierten Funktionen. Als Beispiele für OLE Standardschnittstellen vergleichen sie die Einträge über *IDispatch* und *IUnknown*.

  Außer den Standardschnittstellen kann eine Objekt seine eigenen *speziell erstellten* Schnittstellen implementieren. Natürlich kann das System Funktionen aus speziell erstellten Schnittstellen nicht aufrufen, da es

von deren Existenz nichts weiß. Aber andere »informierte« Anwendungen können diese benutzen. ObjectComponents arbeitet intern durch Hilfe einer Schnittstelle, das von Borland speziell erstellt wurde. S. *BOCOLE Support Library*.

ObjectComponents erspart es Ihnen, besondere Schnittstellen zu verstehen oder zu implementieren. Fortgeschrittene Benutzer, die die Schnittstellen direkt manipulieren oder mit Ihren eigenen mischen möchten, können dies natürlich nach Belieben tun.

- **IUnknown Interface:** Die Grund-Schnittstelle, die alle OLE-Objekte implementieren muß. Mit den drei Arbeitweisen der Schnittstelle *IUnknown* können Sie Objekte nach unterstützten Zeigern fragen, jedes Objekt nach ebenso unterstützten Zeigern zu einer anderen Schnittstelle fragen und den Referenzzähler des Objekts anpassen. Weitere Informationen befinden sich in der Hilfedatei OLE2HELP.HLP.
- **Lokalisierung:** (Localization) Anpassung einer Anwendung, um Zeichenketten in der Benutzersprache darzustellen, welche Sprache das auch sein mag. OLE-Server müssen die Sprache Ihrer Client-Programme sprechen. Wenn ein Automatisierungs-Server in mehreren Ländern vermarktet wird, dann müssen die Befehle in den verschiedenen Sprachen erkannt werden. Ein Verknüpfungs- oder Einbettungs-Server registriert Zeichenketten, die seine Objekte dem Benutzer beschreiben, und auch diese sollten in mehreren Sprachen erhältlich sein, um jeder vom Benutzer verlangten Sprache gerecht zu werden. ObjectComponents läßt Sie Übersetzungen für alle Ihre Zeichenketten als Ressource XFLAT in Ihrer Ressourcendatei plazieren. ObjectComponents wählt die richtige Zeichenkette zur richtigen Zeit.
- **ObjectComponents Framework:** Ein Satz von C++ Klassen von Borland International, die sowohl Verknüpfungs- und Einbettungsfunktionen, als auch Automatisierungsfunktionen einschließen. Intern implementieren die ObjectComponents-Klassen Standardschnittstellen und speziell erstellte Schnittstellen. Mit ObjectComponents schreiben Sie für OLE unter Verwendung von bekannten Programmiermodellen wie Vererbung oder Fensterbotschaften, anstatt COM-Schnittstellen zu implementieren.
- **ObjectWindows-Bibliothek:** Eine Menge von C++ Klassen von Borland International, die Standardfunktionen der Windows Programmierung wie z.B. die Verwaltung von Fenstern und Dialogfenstern einschließen. Die aktuelle Version von ObjectWindows führt einige neue Klassen ein, wie z.B. *TOleWindows* und *TOleView*, die ObjectComponents-Klassen dazu verwenden, OLE-Fähigkeiten zu erlangen. Die neuen Klassen erleichtern das Hinzufügen von OLE-Unterstützung zu existierenden ObjectWindows-Anwendungen.

- **Offenes Bearbeiten:** Bearbeiten eines OLE-Objekts im Fenster des Servers. Offenes Bearbeiten wird duch Aufruf der Aktionsart Open (Öffnen) erreicht. Während offen editiert wird, wird das Server-Fenster vor dem Container-Fenster geöffnet. Wenn die Editiertsitzung beendet ist, dann verschwindet das Server-Fenster und die Änderungen werden im Container-Fenster sichtbar. In Abbildung 19.9 wird offenes Bearbeiten dargestellt. Vergleichen Sie auch *direkte Bearbeitung im Containerdokument*.
- **OLE:** (Object Linking and Embedding) Eine Erweiterung des Windows Systems. (In den aktuelleren Windows Versionen ist OLE ein integrierter Teil des Systems, und keine Erweiterung mehr.) Die neuen von OLE implementierten Befehle und die von ihm definierten Schnittstellen fügen dem System drei neue Merkmale hinzu: Verknüpfen und Einbetten, Automatisierung und E/A von zusammengesetzten Dateien.
- **OLE-Schnittstelle:** Siehe *Schnittstelle*.
- **Referenzzählung:** Die Verwaltung eines Zählers, der festhält, wieviele Clients ein Objekt besitzt. Jeder Programmabschnitt, der verlangt, daß das Objekt existiert, ruft die Methode *AddRef* des Objekts auf, um den Referenzzähler zu erhöhen. Wenn das Client-Programm fertig ist, wird die Methode *Release* aufgerufen, um den Referenzzähler zu erniedrigen. Wenn der Referenzzähler durch Aufruf von *Release* Null erreicht, dann wird dem Objekt erlaubt, sich selbst zu zerstören.

  Jedes OLE-Objekt besitzt die Methoden *AddRef* und *Release*, weil sie Teil der Schnittstelle *IUnknown* sind. Zu wissen, wer ein Client ist, um *AddRef* oder *Release* aufzurufen, kann manchmal sehr kompliziert sein. ObjectComponents verwaltet die Referenzzählung für Sie. Nur für fortgeschrittene Benutzer wird es nötig sein, *AddRef* und *Release* direkt aufzurufen.
- **Registrierungsobjekt:** Ein Objekt mit Typ TRegistrar oder TOcRegistrar. Jede ObjectComponents-Anwendung braucht ein Registerobjekt. Das Register verarbeitet die Kommandozeile der Anwendung, setzt Laufmodus-Flags, verifiziert die Anwendungseinträge in der Registrierungsdatenbank und ruft die Factory-Funktionen der Anwendung auf, um die Anwendung zu starten.
- **Registrierung:** Gibt Informationen über die Anwendung an das System weiter. OLE-Programme führen zwei Arten der Registrierung durch. Wenn eine Anwendung zum ersten Mal installiert wird, schreibt ObjectComponents Informationen aus den Registrierungstabellen der Anwendung in die Systemdatenbank. Diese Information ist statisch und muß nur einmal aufgezeichnet werden. Das Registerobjekt erfüllt diese Aufgabe.

Bei nachfolgenden Starts der Anwendung durch den Benutzer sagt ObjectComponents OLE, daß die Anwendung läuft und es registriert ein *Factory* für jeden Dokumenttyp, den die Anwendung produziert. Wenn die Anwendung endet, dann löscht ObjectComponents die Registrierung der *Factory*. Die Objekte *TOcApp* und *TRegistrar* erfüllen diese Aufgaben.

- **Registrierungsdatenbank:** siehe System-Registrierungsdatenbank
- **Registrierungstabelle:** Eine durch Registrierungsmakros aufgebaute Tabelle mit Informationen über eine Anwendung, oder die von einer Anwendung erzeugten Dokumenttypen. Die Makros erzeugen eine Struktur des Typs *TRegList*. Das Registrierungsobjekt liest die Registrierungsstruktur und kopiert die nötigen Informationen in die System-Registrierungsdatenbank.
- **Server:** Eine für andere Anwendung Objekte erzeugende Anwendung. OLE unterstützt zwei Arten von Servern: verknüpfende und einbettende Server und Automatisierungs-Server. Ein verknüpfender und einbettender Server erzeugt Datenobjekte, die von Containern in ihre zusammengesetzten Dokumente eingefügt, fallengelassen oder hinzugefügt werden können. Ein Automatisierungs-Server erzeugt Objekte, die durch andere Anwendungen durch Aussenden von auszuführenden Befehlen manipuliert werden können. (Eine einzelne Anwendung kann auf Wunsch beide Arten von Objekten erzeugen. Es ist sogar möglich, automatisierte Objekte zu verknüpfen oder einzubetten.)
- **System-Registrierungsdatenbank:** Eine strukturierte Informationsablage über auf einem bestimmten Computer installierte Anwendungen. Im 16-Bit Windows liegt die Datenbank in der Datei REG.DAT. Im 32-Bit Windows heißt die Datenbank die *Systemregistrierung* (System Registry) und liegt in eigenen Systemdateien. Anwendungen tragen ihre Informationen während der Installation ein. Diese Informationen enthalten Kennungen der Anwendung und ihre Dokumente, Beschreibungen der Anwendung und ihrer Dokumente, Pfade zu den Anwendungsdateien, die voreingestellten Erweiterungen der Dokumentendateien der Anwendung und andere Details, die dem OLE-System helfen, Server mit ihren Objekten in Verbindung zu bringen.
- **Typenbibliotheken:** Eine Datei, die die Befehle beschreibt, die ein Automatisierungs-Controller unterstützt. Eine Typenbibliothek zu erzeugen, ist für einen Automatisierungs-Server der Standardweg, um die zu implementierende Programm-Schnittstelle zu veröffentlichen. Die Typenbibliothek erklärt, was für Objekte von dem Server erzeugt werden, und beschreibt die Eigenschaften und Arbeitsweisen des Objekts. Die Typeninformation wird von den Compilern und Interpretierern gelesen, die die Automatisierungsbefehle verarbeiten. Einige Anwendungen erlauben dem Benutzer außerdem, die Typeninformationen zu

durchlaufen. Jeder ObjectComponents-Automatisierungs-Server generiert eine Typenbibliothek, sobald sie mit dem -TypeLib Kommandozeilenschalter aufgerufen wird. Typenbibliotheken verwenden normalerweise die Erweiterung .TLB. Ein Automatisierungs-Server registriert den Ort seiner Typenbibliothek während der Installation.

- **Aktionsart:** Ein Befehl, den ein Verknüpfungs- und Einbettungs-Server mit seinen Objekten ausführen kann. Der Server sagt dem Container, was durch eine Aktionsart unterstützt wird, und der Container stellt die Zeichenkette für die Aktionsart in seinem Menü *Bearbeiten* dar. Um eine Aktionsart auszuführen, wählt der Benutzer ein Objekt und die gewünschte Aktionsart im Menü aus. Der Container aktualisiert das Menü *Aktionsarten*, sobald der Benutzer eine neue Aktionsart auswählt.

  Der Server kann jede beliebige Aktionsart unterstützen. Die meisten Server unterstützen die Aktionsarten Edit(Bearbeiten) und Open (Öffnen), um offen oder direkt zu bearbeiten. In Abhängigkeit von den enthaltenenen Daten kann ein Server andere Daten wie z.B. Play (Abspielen) und Rewind unterstützen.

- **Teil:** (Part) Ein in ein zusammengesetztes Dokument verbundenes oder eingebettetes Objekt. Ein ObjectComponents-Container erzeugt ein Objekt der Klasse *TOcPart*, um jedes Objekt verbunden oder eingebettet in seinem Dokument darzustellen.

  Teil ist das vom Container benutzte Wort für ein durch einen Server erzeugtes Objekt. Im Programm des Servers wird dasselbe Objekt als normales Server-Dokument erzeugt. ObjectComponents präsentiert OLE das Dokument als OLE-Objekt. Der Container erzeugt *TOcPart*, sobald er das OLE-Objekt erhält. Wenn das Teil gezeichnet werden soll, dann kommuniziert das Teilobjekt durch OLE mit dem Ausschnittsobjekt des Servers.

- **Verbinderobjekt:** Eine mit OLE kommunizierende Klasse von ObjectComponents. Verbinderobjekte verbinden Teile Ihrer Anwendung mit OLE. *TOcApp* führt z.B. OLE-Funktionen für die Anwendung aus. *TOcView* führt OLE-Funktionen für einen Ausschnitt eines Dokuments aus. *TOcPart* führt OLE-Funktionen für ein verbundenes oder eingebettetes Objekt aus. Die Verbinderobjekte sind mit den entsprechenden Teilen Ihrer Anwendung zusammenarbeitende Partner. Sie rufen deren Methoden auf, und sie schicken Ihnen Botschaften zu. Verbinder sind Component Object Model (COM) Objekte und implementieren COM-Schnittstellen. (Nicht alle Klassen von ObjectComponents sind Verbinder.)

- **Verknüpfungsquelle:** (Link Source) Jenes Dokument, auf das sich eine Verknüpfung bezieht, die Quelle für die Daten in einem verknüpfenden Objekt. Normalerweise ist die Verknüpfungsquelle ein Server-Dokument, es ist aber nicht ungewöhnlich, daß ein Container Quellver-

knüpfungsdaten exportiert, damit andere Anwendungen mit im Container-Dokument eingebetteten oder verknüpften Objekten verbunden werden können. Informationen darüber, wie eine Verknüpfungsquelle entsteht, finden Sie im Eintrag für REGFORMAT im *ObjectWindows Referenzhandbuch*.

- **Verknüpfende Objekte:** (Linked Objects) Ein in einem Container-Dokument erscheinendes Objekt, dessen Daten aber eigentlich in einer anderen Datei stehen. Wenn ein Objekt in einen Container eingefügt oder gezogen wird, dann kann der Benutzer auf Wunsch eine *Verknüpfung* zu diesem Objekt herstellen, anstatt es einzubetten. Der Container erhält oder speichert die Daten des verknüpften Objekts nicht in seinem eigenen Dokument. Statt dessen erhält er nur eine Zeichenkette, die die eigentlichen Daten enthaltende Datei identifiziert.

  Mehrere Container können mit demselben Objekt verknüpfen werden. In diesem Fall erhalten alle Objekte dieselbe Zeichenkette, die auf dieselbe Datei zeigt. Wenn die Daten im ursprünglichen Objekt verändert werden, dann werden die Veränderungen automatisch in allen Dokumenten, mit denen es verknüpften ist, aktualisiert. Wenn der Benutzer ein Objekt in mehreren Containern *einbettet*, dann besitzt jeder Container eine eigene Kopie der Objektdaten, und Änderungen in einer Kopie bewirken in den anderen Kopien gar nichts.

- **Zusammengesetzte Datei** Eine einzelne Datei, die vom Betriebssystem in voneinander unabhängige Teile, genannt *Lager* (Storages), unterteilt werden. Jedes Lager besitzt dabei seinen eigenen E/A Dateizeiger, damit ohne Aufrechterhaltung von Abzweigungen zu anderen Lagern Daten in jedem beliebigen Lager gelesen, geschrieben, überschrieben und gelöscht werden können. Es ist sehr nützlich, zusammengesetzte Dokumente in zusammengesetzten Dateien abzulegen, da für jedes verbundene oder eingebettete Objekt ein neues Lager erzeugt werden kann. OLE erweitert durch die Implementation von Schnittstellen das Dateisystem, um zusammengesetzte Dateien zu unterstützten.

- **Zusammengesetztes Dokument:** *Ein Dokument, das durch andere Anwendungen eingebrachte OLE-Objekte, enthält. Ein zusammengesetztes Dokument könnte z.B. Teilinformationen eines Arbeitsblatts, einer Datenbank oder einer Textverarbeitung in einem einzigen Dokument enthalten, damit der Benutzer mit einem einzigen Befehl speichern oder laden kann. Die Objekte aus anderen Anwendungen sind entweder im Containerdokument verbunden oder eingebettet worden.*

K A P I T E L 20

Erzeugen eines OLE-Containers

Ein OLE-Container ist eine Anwendung, die in ihren eigenen Dokumenten Datenobjekte aus anderen Anwendungen speichern kann. Ein Container kann Objekte linken oder sie in seine eigenen Objekte einbetten. Ein Programm, das Objekte zum Linken oder Einbetten erzeugt, wird Server genannt. Dieses Kapitel erklärt, wie man vorhandene Programme in OLE-Container umwandeln kann. Es beschreibt für drei verschiedene Arten von Programmen die für die Anpassung nötigen Schritte:

- Eine ObjectWindows-Anwendung, die das Dok/View-Modell verwendet
- Eine ObjectWindows-Anwendung, die das Dok/View-Modell nicht verwendet
- Eine C++ Anwendung, die ObjectWindows nicht verwendet

Der erste Fall ist sehr einfach. Der letzte Fall bezieht sich ganz auf den ObjectComponents Framework und benötigt den meisten neuen Code. Aber das ist immer noch wesentlich einfacher, als direkt in OLE zu programmieren.

Eine Dok/View-Anwendung in einen OLE-Container umwandeln

Es sind nur wenige Änderungen nötig, um eine Dok/View-Anwendung in einen OLE-Container umzuwandeln. Die folgende Aufzählung beschreibt kurz die Änderungen. In den nachfolgenden Abschnitten werden die jeweiligen Änderungen genauer beschrieben.

1. Verknüpfen Sie Ihre Anwendungen, Fenster, Dokumente und View-Objekte mit OLE.
   - Leiten Sie Ihre Anwendungsklassen sowohl von *TOcModule* als auch von *TApplication* ab.

- Leiten Sie Rahmenfenster, Dokumente und View-Klassen von neuen OLE-fähigen Klassen ab.
- Erzeugen Sie ein TAppDictionary-Object.

2. Registrieren der Anwendung
   - Verwenden von Makros, die Registrierungstabellen zur Beschreibung Ihrer Anwendung anfertigen
   - Erzeugen eines Registrar-Objektes und Aufrufen seiner Methode *Run*
3. OLE-Kommandos unterstützen
   - Initialisieren Ihres Editormenüs und Ihrer Mauspalette, damit sie die zweckmäßigen vordefinierten Bezeichner zur Unterstützung von OLE-Standardkommandos verwenden.
   - Befähigen Ihrer Öffnen- und Speichern-Kommandos zum Lesen und Schreiben von eingebetteten Objekten in Ihren Verbunddokumenten.
4. Erstellen der Container-Anwendung
   - Einbinden neuer OLE-Header von ObjectWindows am Anfang Ihres Quell-Codes
   - Compilieren des Programms mit dem Speichermodell **large**. Linken mit den Bibliotheken von OLE und ObjectComponents

Das ist alles, was Sie tum müssen. Mit Hilfe dieser Schritte können Sie einen OLE-Container erzeugen, der alle folgenden Merkmale besitzt.

- Linken
- Operationen von OLE mit der Zwischenablage
- In-Place-Bearbeiten
- Verbunddokument-Speicherabschnitt
- Einbetten
- Drag&Drop-Operationen
- Zusammenführen von Mauspalette und Menü

Sie können auch Standardmerkmale der Benutzerschnittstelle von OLE 2 verwenden, wie z.B. Aktionsarten von Objekten im Menü *Bearbeiten*, das Dialogfeld *Objekt einfügen* und ein Pop-Up-Menü, das erscheint, wenn der Benutzer mit der rechten Maustaste auf ein eingebettetes Objekt klickt.

ObjectComponents stellt ein voreingestelltes Verhalten für alle diese allgemeinen Merkmale von OLE zur Verfügung. Wenn Sie das voreingestellte Verhalten abändern wollen, können Sie sich dafür entscheiden, die voreingestellte Ereignisbehandlung für von ObjectComponents gesendete Nachrichten zu überschreiben. Eine Auflistung der Ereignismeldungen finden Sie in den Tabellen 19.5 und 19.6.

Die Beispielcodes in diesen Abschnitten basieren auf den Beispielprogrammen *STEP14.CPP* und *STEP14DV.CPP* in *EXAMPLES/OWL/TUTORIAL*. Dort sehen Sie ein vollständiges und lauffähiges Programm, das alle die beschriebenen Schritte beinhaltet.

1. Objekte mit OLE verknüpfen

Ihre Anwendungen, Fenster, Dokumente und View-Objekte müssen von der neuen OLE-fähigen Klassen Gebrauch machen. Der Konstruktor für die Anwendungs-Objekte erwartet ein Anwendungs-Dictionary-Object. Erzeugen Sie dieses deshalb zuerst.

Ableiten eines Anwendungsobjekts von TOcModule

Das Anwendungsobjekt eines ObjectComponents-Programms muß von *TOcModule* und von *TApplication* abgeleitet sein. *TOcModule* koordiniert einige grundlegenden Arbeiten, die sich auf die Registrierung und die Speicherverwaltung beziehen. Sie verbindet außerdem Ihre Anwendung mit OLE. Genauer gesagt, verwaltet *TOcModule* das Verbindungsobjekt, das die COM-Schnittstelle im Auftrag einer Anwendung bildet.

Wenn die Deklaration Ihres Anwendungsobjekts etwa so aussieht:

```
class TMyApp : public TApplication {
  public:
    TMyApp() : TApplication(){};

};
```

ändern Sie es etwa so:

```
class TMyApp : public TApplication, public TOcModule {
  public:
    TMyApp(): TApplication(::AppReg["description"]){};

};
```

Der Konstruktor der überarbeiteten Klasse *TMyApp* übergibt an einen in der Variable *AppReg* gespeicherten String aus der Registrierungstabelle an *TApplication*. Die Registrierungstabelle wird später im Abschnitt »Registrierung des Containers« dargestellt.

Ableiten von OLE-Klassen

ObjectWindows enthält Klassen, die Fenster, Dokumente und Views mit den Klassen von ObjectComponents interagieren lassen. Die OLE-Klassen von ObjectWindows enhalten voreingestellte Implementierungen für die normalen OLE-Operationen. Um ein vorhandenes ObjectWindows-Programm an OLE anzupassen, ändern Sie seine abgeleiteten Klassen so, daß sie von den OLE-Klassen erben. Tabelle 20.1 zeigt, wie jede Nicht-OLE-Klasse durch eine OLE-Klasse ersetzt werden kann.

Tabelle 20.1: Nicht-OLE-Klassen und zugehörige Klassen, die OLE-Unterstützung hinzufügen

Nicht-OLE Klassen	OLE Klassen
TFrameWindow	TOleFrame
TMDIFrame	TOleMDIFrame
TDecoratedFrame	TOleFrame
TDecoratedMDIFrame	TOleMDIFrame
TWindow	TOleWindow
TDocument	TOleDocument
TView	TOleView
TFileDocument	TOleDocument

Die Klassen TOleFrame und *TOleMDIFrame* erben beide von Dekorations-Fensterklassen. Die Benutzerschnittstelle von OLE 2 benötigt Container für die Behandlung von Mauspaletten und Statuszeilen. Selbst wenn der Container nicht über Dekorationen verfügt, müssen sich die Server selbst im Fenster des Containers darstellen. Die Fensterklassen von OLE übernehmen das für Sie.

Immer wenn ihr vorhandenes OWL-Programm eine Nicht-OLE-Klasse verwendet, ersetzen Sie sie auf die hier gezeigte Weise mit einer OLE-Klasse. Die Änderung ist mit fetter Schrift markiert.

Before:

```
// Deklaretion einer Fensterklasse vor OLE
class TMyFrame: public TFrameWindow    { /* Declarationen */ );
```

After:

```
// Neue Deklaration derselben Fensterklasse
class TMyFrame: public TOleFrame       { /* Declarationen */ );
```

Hinweis

Wenn Ihre Implementierung einer Klasse direkte Aufrufe ihrer Basisklasse durchführt, dann stellen Sie auch die Änderungen der Basisklassenaufrufe sicher. Antworttabellen beziehen sich auch auf die Basisklassen und müssen aktualisiert werden.

Erzeugen eines Anwendungsverzeichnisses

Eine Anwendungsverzeichnis enthält Informationen über den gegenwärtig aktiven Prozeß. Das ist bei DLLs besonders nützlich. Wenn mehrere Prozesse eine DLL gleichzeitig verwenden, muß die DLL mehrere Kopien der globalen, statischen und dynamischen Variablen enthalten, die den aktuellen Status jedes Prozesses darstellen. Die DLL-Version von ObjectWindows enthält beispielsweise ein Verzeichnis mit dem sie die *TApplication* wiederbekommt, die zu dem gerade aktiven Client-Prozeß gehört.Wenn Sie einen ausführbaren Server in einen DLL-Server umwandeln, muß Ihre Anwendung auch ein Verzeichnis des *TApplication*-Objekts enthalten, das jeden seiner Container-Klienten darstellt. Wenn Ihre DLL eine DLL-Version von ObjectWindows verwendet, dann benötigt

Ihre DLL sein eigenes Verzeichnis und kann nicht das aus ObjectWindows verwenden.

Das Makro DEFINE_APP_DICTIONARY stellt einen einfachen und einheitlichen Weg dar, um Anwendungsobjekte einer beliebigen Anwendung zu erzeugen, unabhängig davon, ob es sich um einen Server, eine DLL oder eine ausführbare Datei handelt. Fügen Sie diese Anweisung bei Ihren anderen statischen Variablen ein.

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

Für jede zur statischen Version der DLL gelinkten Anwendung erzeugt das Makro einfach eine Referenz auf das Anwendungsverzeichnis in ObjectWindows. Für DLL-Server, die die DLL-Version von ObjectWindows benutzen, wird jedoch eine Instanz der Klasse *TAppDictionary* erzeugt.

Hinweis

Nennen Sie Ihr Verzeichnisobjekt *AppDictionary*, um von Factory Templates wie *TOleDocViewFactory* (im Abschnitt "*Registrierung eines Containers*" erläutert) profitieren zu können.

2. Registrierung eines Containers

Zur Registrierung Ihrer Anwendung mit OLE erzeugen Sie Registrierungstabellen, die die Anwendungen und die Art der von jenen erzeugten Dokumente beschreiben. Erzeugen Sie ein Registrar-Objekt, um die Information in den Tabellen zu produzieren.

Erzeugen von Registrierungstabellen

OLE benötigt Programme, die sich selbst durch die Registrierung einheitlicher Bezeichner und Namen identifizieren. OLE muß ebenfalls wissen, was für Zwischenablageformate ein Programm unterstützt. Dok/View-Anwendungen registrieren auch die Dateierweiterungen und Dokumentenflags. Um die vielen neuen Elemente zu bewältigen, die eine Anwendung möglicherweise registrieren könnte, können Sie in ObjectWindows 2.5 Makros zum Erstellen von Elementen beinhaltenden Strukturen verwenden. Dann können Sie die Struktur an da Objekt übergeben, daß die Information benötigt. Der Vorteil dieser Methode liegt in der Flexibilität der Struktur. Sie kann so viele neue Elemente enthalten, wie Sie benötigen.

Hinweis

Frühere Versionen von ObjectWindows übergaben einige derselben Informationen über Parameter. Der alte Code funktioniert noch unverändert, aber die Übergabe von Informationen in Registrierungsstrukturen ist die gegebene Methode für alle neuen Anwendungen. Ein Dok/View-Container füllt eine Struktur mit Informationen über die Anwendung aus und erzeugt dann eine andere um jede seiner Dok/View-Paare zu beschreiben. Im nächsten Abschnitt werden Sie sehen, wie die Struktur mir der Anwendungsinformation an den Konstruktor von *TOcApp* übergeben werden. Die Strukturen zur Dokumentregistrierung werden an der Template-Konstruktor des Dokumentes übergeben.

Hier sind die Kommandos, die einen typischen Container registrieren:

```
REGISTRATION_FORMAT_BUFFER(100)      // Ermöglicht weitere
                                     // Speicherreservierung für
                                     // expandierende Makros

BEGIN_REGISTRATION(AppReg)           // Information für den Konstruktor
                                     // von  TOcRegistrar
  REGDATA(clsid,       "{383882A1-8ABC-101B-A23B-CE4E85D07ED2}")
  REGDATA(description,"Drawing Pad Container")
END_REGISTRATION

BEGIN_REGISTRATION(DocReg)           // Information für das
Dokument-Template
  REGDATA(progid,      "DrawPad.Document.14")
  REGDATA(description,"Drawing Pad (Step14--Container)")
  REGDATA(extension,   "p14")
  REGDATA(docfilter,   "*.p14")

REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRe
gisterExt)
  REGFORMAT(0, ocrEmbedSource,  ocrContent,  ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent,  ocrMfPict|ocrStaticMed, ocrGet)
  REGFORMAT(2, ocrBitmap, ocrContent,  ocrGDI|ocrStaticMed, ocrGet)
  REGFORMAT(3, ocrDib, ocrContent,  ocrHGlobal|ocrStaticMed, ocrGet)
  REGFORMAT(4, ocrLinkSource, ocrContent,  ocrIStream, ocrGet)
END_REGISTRATION
```

Die Registrierungsmakros erzeugen Strukturen des Typs *TRegList*. Jeder Eintrag in einer Registrierungsstruktur enthält einen Schlüssel wie *clsid* oder *progid*, sowie einen dem Schlüssel zugewiesenen Wert. Die Object-Components finden die Werte intern durch das Suchen nach diesen Schlüsseln. Die Reihenfolge, in der die Schlüssel erscheinen, spielt keine Rollen. Fügen Sie nach der Deklaration des Anwendungsverzeichnisses ein Registrierungsmakro ein. Obwohl der Wert des *clsid*-Schlüssels eine einzelne Nummer ist, die Ihre Anwendung identifiziert, empfehlen wir Ihnen, einen neuen Wert zur Verwendung des Hilfsmittels GUIDGEN.EXE zu generieren. (Das *ObjectWindows Referenzhandbuch* erläutert für *clsid* andere Wege, um einen Bezeichner zu generieren.) Natürlich wird der Wert des Schlüssels *description* verändert , um Ihren Container zu beschreiben. Das Beispiel erzeugt zwei Strukturen, eine heißt *AppReg* und die andere *DocReg*. Der Satz von Schlüsseln, den Sie in die Struktur legen, hängt davon ab, welche Fähigkeiten von OLE Sie unterstützen wollen und ob die Struktur Anwendungsinformation oder Dokumentinformation enthält. Die Makros im Beispiel zeigen die minimalen Informationen, die en Container enthalten sollte.

Tabelle 20.2 beschreibt all die Registrierungsschlüssel, die ein Container verwenden kann. Sie zeigt, welche wahlfrei sind und welche benötigt

werden, welche zur Registrierungstabelle der Anwendung und welche zur Registrierungstabelle des Dokumentes gehören.

Tabelle 20.2: Schlüssel, den ein Container registriert, um Linken und Einbetten zu unterstützen (Fortsetzung)

Schlüssel	In AppReg?	In DocReg?	Beschreibung
clsid	Ja	Wahlfrei	Globally unique identifier (GUID); wird automatisch für die Struktur *DocReg* erzeugt.
Beschreibung	Ja	Ja	Beschreibender String (bis zu 40 Zeivhen)
progid	Nein	Ja für eine gelinkte Quelle	Bezeichner für ein Programm oder Dokumententyp (einmaliger String)
extension	Nein	Wahlfrei	Dokumentdatei-Erweiterung verbunden mit einem Server
docfilter	Nein	Ja	Platzhalter-Dateifilter für das Dialogfeld "*Datei/Öffnen*"
docflags	Nein	Ja	Optionen für die Ausführung des Dialogfeldes "*Datei/Öffnen*"
formatn	Nein	Ja	Ein Zwischenablageformat, das der Container unterstützt
directory	Nein	Wahlfrei	Voreingestelltes Verzeichnis für die Speicherung von Dokumentdateien
permid	Nein	Wahlfrei	Namens-String ohne Versionsinformationen
permname	Nein	Wahlfrei	Beschreibender String ohne Versionsinformation
version	Wahlfrei	Nein	Größere oder kleinere Versionsnummern (Voreinstellung ist "1.0")

Die Tabelle zeigt, was für Container-Dokumente benötigt wird, die Linken oder Einbetten unterstützen. Für Dokumente, die keines von beiden unterstützen, benötigt der Container lediglich *docflags* and *docfilter*.

Wenn Ihr Container auch ein Linken- und Einbetten-Server oder ein Automatisierungs-Server ist, dann sollten Sie die Server- oder die Automatisierungstabelle in diesem Handbuch zu Rate ziehen. Registrieren Sie alle benötigten Schlüssel in einer der Tabellen, die zu Ihrer Anwendung gehören. Weitere Informationen über Registrierungstabellen erhalten Sie im Einführungsabschnitt über Registrierung in diesem Handbuch. Weitere Informationen über individuelle Registrierungsschlüssel und ihre Werte erhalten Sie im *ObjectWindows Referenzhandbuch.*

Die den Schlüsseln zugewiesenen Werte können in entsprechende Spracheinstellungen des Systems übersetzt werden. Weitere Informationen über Lokalisierung befinden sich in den Abschnitten »Registrierung ortsbezogener Einträge« und »Ortsbezogene Symbolnamen«.

Einführung in Registrierungsmakros

Das erste Makro im Beispiel heißt REGISTRATION_FORMAT_BUFFER und setzt die Größe temporät benötigten Puffers so, daß die folgenden Makros erweitert werden.

Die Makros REGDATA, REGFORMAT und REGDOCFLAGS plazieren Einträge in der Registrierungsstruktur. Alle Registrierungsmakros sind im *ObjectWindows-Referenzhandbuch* dokumentiert.

Der erste Parameter von REGDATA ist ein Schlüssel und der zweite ist ein mit dem Schlüssel verbundener Wert. Im Beispiel beginnt die Struktur *AppReg* damit, dem Schlüssel *clsid* einen Wert zuzuweisen. *clsid* ist ein *globally unique identifier* (GUID), der die Anwendung beschreibt. Der Text-String *progid* der Anwendung dient demselben Zweck. Der Schlüssel *description* beschreibt kurz die Anwendung. Von diesen drei Schlüsseln ist nur der Wert *description* für den Benutzer sichtbar (Wenn die Anwendung automatisiert ist, sehen die Benutzer ebenfalls *progid*, siehe Kapitel 22). Die Struktur eines Dokuments registriert seine eigenen *progid* and *description*. Jedes Dokument benötigt sein eigenes eindeutiges *clsid*, selbst wenn Sie es weglassen. Die ObjectComponents unterstützen das für Sie durch die Inkrementierung des *clsid* einer Anwendung.

REGFORMAT-Einträge listen die Datenformate, die ein Container in die Zwischenablage legen kann. Der erste Parameter setzt eine Prioritätsanordnung für die von Ihnen benutzten Formate. *0* markiert das Format mit der höchsten Genauigkeit und höhere Nummern zeigen niedrigere Genauigkeiten an. Der zweite Parameter stellt ein Datenformat dar. Der andere Parameter gibt an, welchen Darstellungaspekt eines Formates Sie benutzen, welches Medium Sie für den Datentransfer verwenden und ob Sie in diesem Format Daten in oder aus der Zwischenablage nehmen können. Alle weiteren Daten, die Sie mit REGFORMAT angeben, sind mit der Windows-Zwischenablage für Sie gespeichert.

Selbst ein einfacher Container ist fähig, OLE-Objekte in die Zwischenablage zu lagen. Wenn der Benutzer ein Gelinktes oder Eingebettetes Objekt aus dem Container-Dokument wählt, und dieses über die Zwischenablage zu einem anderen Container übertragen will, dann muß der erste Container als ein Server fungieren, indem er zumindest die Formate *ocrEmbedSource* oder *ocrLinkSource* unterstützt. Jede Anwendung, die eines dieser Formate registriert, muß auch *ocrMetafilePict* registrieren. Gewöhnlich werden die fünf Formate registriert, die im Beispiel gezeigt werden. ObjectComponents bearbeitet OLE-Objekte automatisch in jedem beliebigen Standard-Format. Sie brauchen nur diejenigen zu registrieren, die sie unterstützt haben wollen.

Sie müssen den Datenformatparameter mit einem String, der ihren Formatnamen enthält, ersetzen, um benutzerdefinierte Formate zu registrieren.

```
REGFORMAT(3, "MyOwnFormat", ocrContent, ocrIStorage, ocrGet)
```

Wenn Sie Zwischenablageformate von Clients registrieren, müssen Sie OLE auch Strings zur Verfügung stellen, die Ihr Format für Dialogfenster angeben. Rufen Sie dazu *AddUserFormatName* auf, eine von *TOleFrame* abgeleitete Klassenmethode.

Mehr Informationen zu diesem Thema finden Sie im Abschnitt »REGFORMAT« im *ObjectWindows Reference Guide*.

REGDOCFLAGS fügt der Registrierungsstruktur einen Eintrag hinzu, der Flags für eine Dokumentvorlage enthält. Die Flags geben die Optionen für das allgemeine Dialogfenster *Datei öffnen* an.

Nach der Erzeugung von Registrierungstabellen müssen Sie diese an die entsprechenden Objektkonstruktoren übergeben. Die Struktur *AppReg* wird an den Konstruktor von *TOcRegistrar* übergeben wie im Abschnitt »Erzeugen eines Registrar-Objekts« beschrieben. In einer Dok/View-Anwendung werden Dokumentregistrierungstabellen an den Dokumentvorlagenkonstruktor übergeben.

```
DEFINE_DOC_TEMPLATE_CLASS(TMyOleDocument, TMyOleView, MyTemplate);
MyTemplate myTpl(DocReg);
```

Ein Programm, das mehrere Dokumentvorlagen verwendet, sollte so eingerichtet werden, daß es für jede Vorlage eine andere Registrierungsstruktur erzeugt. Jede Registrierungsstruktur muß mit dem Makro BEGIN_REGISTRATION beginnen und einen anderen Namen haben, z.B. *DocReg1* und *DocReg2*.

Die gesamte Information, die normalerweise an einen Dokumentvorlagenkonstruktor übergeben wird, kann in einer Registrierungsstruktur angegeben werden, unter Verwendung von REGFORMAT, REGDOCFLAGS und REGDATA. Frühere Versionen von OWL übergaben der Dokumentvorlage diese Information als eine Reihe einzelner Parameter. Die alte Methode wird immer noch unterstützt, um Rückwärtskompatibilität zu gewährleisten, sollte aber für neue Programme, ob sie nun OLE verwenden oder nicht, zugunsten der Verwendung von Registrierungsmakros aufgegeben werden, wenn Dokumentvorlagenparameter angegeben werden müssen.

Erzeugen eines Registrar-Objekts

Jede ObjectComponents-Anwendung benötigt ein Registrar-Objekt, um ihre Registrierungsaufgaben zu verwalten. In einer OLE-Anwendung ist der Registrar ein Objekt des Typs *TOcRegistrar*. Am Anfang Ihrer Quell-Code-Datei müssen Sie eine globale Variable deklarieren, die einen Zeiger auf den Registrar enthält.

```
static TPointer<TOcRegistrar> Registrar;
```

Die Template *TPointer* stellt sicher, daß die Instanz von *TOcRegistrar*- gelöscht wird, wenn das Programm endet.

Hinweis

Nennen Sie die Variable *Registrar*, um von der Factory Callback Template zu profitieren, die im Konstruktor des Registrars verwendet wird.

Der nächste Schritt besteht darin, Ihre *OwlMain*-Funktion zu ändern, um ein neues *TOcRegistrar*-Object zu alloziieren und den globalen Zeiger *Registrar* zu initiieren. Der Konstruktor von *TOcRegistrar* erwartet drei Parameter: die Registierungsstruktur der Anwendung, den Factory Callback der Komponente und den Kommandozeilen-String, der die Anwendung aufgerufen hat.

- Die Registrierungsstruktur erzeugen Sie mit den Registrierungsmakros.
- Den Factory Callback erzeugen Sie mit einer Template-Klasse. Für eine ObjectWindows-Linken-und-Verknüpfen-Anwendung, die Dok/View verwendet, heißt die Template-Klasse *TOleDocViewFactory*. Der Code im Factory Template setzt voraus, daß Sie ein Anwendungswörterbuch namens *AppDictionary* und einen *TOcRegistrar** namens *Registrar* definiert haben.
- Der Kommandozeilen-String kann aus der *GetCmdLine*-Methode von *TApplication* stammen.

```
int
OwlMain(int /*argc*/, char* /*argv*/ [])
{
  try {
    // Registrar-Objekt erzeugen
    Registrar = new TOcRegistrar(::AppReg, TOleDocViewFactory<TMyApp>(),
                                 TApplication::GetCmdLine());
    return Registrar->Run();
  }
  catch (xmsg& x) {
    ::MessageBox(0, x.why().c_str(), "Exception", MB_OK);
  }
  return -1;
}
```

Nach der Initialisierung des *Registrar*-Zeigers muß Ihre OLE-Container-Anwendung die *Run*-Methode des Registrars anstelle von *TApplication::*Run aufrufen. Im Falle von OLE-Containern ruft die *Run*-Methode des Registrar einfach die *Run*-Methode des Anwendungsobjekts auf, um die Anwendungfenster und Prozessbotschaften zu erzeugen. Die Verwendung der Methode des Registrars macht Ihre Anwendung jedoch OLE-Server-fähig. Der folgende Code zeigt Ihnen ein Beispiel-*OwlMain* vor und nach der Hinzufügung eines Registrar-Objekts. Änderungen sind mit Fettdruck hervorgehoben.

Vor:

```
// Keine OLE-OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  return TMyApp().Run();
}Nach Hinzufügung des Registrar-Objekts:

int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  ::Registrar = new TOcRegistrar(::AppReg,
        TOleDocViewFactory<TMyApp>(),
        TApplication::GetCmdLine());
  return ::Registrar->Run();
}
```

Der letzte Parameter des Konstruktors von *TOcRegistrar* ist der Kommandozeilen-String, der die Anwendung aufruft. Das Registrar-Objekt durchsucht die Kommandozeile nach Schaltern wie *-Embedding* or *-Automation*, die OLE dort eingesetzt haben kann. ObjectComponents führt die Aktionen aus, die von den Schaltern verlangt werden, und entfernt diese danach. Falls Sie aus irgendeinem Grund die OLE-Schalter testen müssen, müssen Sie dies tun, bevor Sie den Registrar konstruieren. Wenn Sie die von OLE eingefügten Schalter nicht benötigen, warten Sie mit dem Bearbeiten der Kommandozeile, bis Sie den Registrar konstruiert haben. Mehr Information über Kommandozeilenschalter finden Sie im Abschnitt »Bearbeiten der Kommandozeile«.

3. Unterstützen von OLE-Kommandos

Ein Container muß einige Standard-OLE-Kommandos in sein *Edit*-Menü eingeben. ObjectWindows implementiert diese Kommandos für Sie. Ein Container muß auch zulassen, daß ObjectComponents jedes gelinkte oder eingebettete Objekt lesen kann, während es Dokumente lädt oder speichert.

Einrichten des Edit-Menüs

Ein OLE-Container gibt OLE-Kommandos in sein *Edit*-Menü ein. Tabelle 20.3 beschreibt die Standard-OLE-Kommandos. Sie brauchen nicht alle zu verwenden, aber jeder Container sollte zumindest Insert Object unterstützen (um dem Benutzer das Einfügen neuer Objekte in ein aktuelles Dokument zu gestatten) und Edit Object (um dem Benutzer die Aktivierung des aktuell markierten Objekts zu gestatten). Die Klasse *TOleView* verfügt über standardmäßige Implementierungen für all diese Kommandos. Sie ruft, wenn nötig, Standard-Dialogfenster auf und bearbeitet die Eingaben des Benutzers. Sie brauchen nur die Kommandos für jede Ansicht, die Sie von *TOleView* ableiten, zum *Edit*-Menü hinzuzufügen.

Tabelle 20.3: Kommandos, die ein OLE-Container in sein Menü *Bearbeiten* eingibt

Menübefehl	Vordefinierter Bezeichner	Kommandobeschreibung
Inhalte einfügen	CM_EDITPASTESPECIAL	Gestattet dem Benutzer, zwischen verschiedenen Formaten zu wählen, um ein Objekt aus der Zwischenablage einzufügen.
Verknüpfung einfügen	CM_EDITPASTELINK	Erzeugt im aktuellen Dokument eine Verknüpfung mit dem Objekt in der Zwischenablage.
Objekt einfügen	CM_EDITINSERTOBJECT	Gestattet dem Benutzer, ein neues Objekt durch Auswahl von einer Liste verfügbarer Typen zu erzeugen.
Verknüpfung bearbeiten	CM_EDITLINKS	Gestattet dem Benutzer, die Liste verknüpfter Objekte im aktuellen Dokument von Hand zu aktualisieren.
Konvertieren	CM_EDITCONVERT	Gestattet dem Benutzer, Objekte von einem Typ zum anderen zu konvertieren.
Objekt bearbeiten	CM_EDITOBJECT	Reserviert einen Platz im Menü für die Aktionsarten des Servers (Aktionen, die der Server am Objekt des Containers ausführen kann).

Wenn Ihr OLE-Server über eine Mauspalette verfügt, weisen Sie ihm den vordefinierten Bezeichner IDW_TOOLBAR zu. ObjectComponents muß in der Lage sein, die Mauspalette des Containers zu finden, falls ein Server seine eigene Mauspalette im Container-Fenster anzeigen will. Falls ObjectComponents die alte Mauspalette identifizieren kann, ersetzt es diese vorübergehend durch eine neue Mauspalette vom Server. Der Container muß IDW_TOOLBAR als seine Fenster-ID verwenden (wie unten gezeigt), damit ObjectComponents die Mauspalette des Containers identifizieren kann.

```
TControlBar *cb = new TControlBar(parent);
cb->Attr.Id = IDW_TOOLBAR;          // diesen Bezeichner verwenden
```

Die Methode *TOleFrame::EvAppBorderSpaceSet* verwendet IDW_TOOLBAR als standardmäßige Implementierung. Ein Container kann seine eigene Implementierung zur Verfügung stellen, um komplexere Situationen, wie das Verbinden mit Mehrfach-Mauspaletten, zu bearbeiten.

Laden und Speichern von Verbunddokumenten

Wenn der Benutzer ein OLE-Objekt in einen Container einfügt oder ablegt, wird das Objekt zu Daten im Container-Dokument. Der Container muß das Objekt zusammen mit dem restlichen Dokument speichern und laden, wann immer der Benutzer die Befehle *Speichern* oder *Öffnen* aus dem *Datei*-Menü wählt. Die neuen Methoden *Commit* und *Open* of *TOleDocument* führen dies für Sie aus. Sie brauchen nur *TOleDocument::Open* in der Implementierung von *Open* aufzurufen, und *TOleDocument ::Com-*

mit in der Implementierung von *Commit*. Der Code, der die eigentlichen Daten Ihres Dokuments liest und schreibt, bleibt davon unberührt.

Da *TOleDocument* von *TStorageDocument* und nicht von *TFileDocument* abgeleitet ist, erzeugt es stets Verbunddateien. *Verbunddateien* sind Features von OLE 2, die den Inhalt einer Datei auf verschiedene Abschnitte verteilen. Sie können jeden Dateiabschnitt lesen oder beschreiben, ohen sich darum zu kümmern, wo auf der Diskette oder auf dem Laufwerk der Abschnitt beginnt oder aufhört. OLE nennt die Abschnitte *Speicherabschnitte*. Die Speicherabschnitte in einer Datei können hierarchisch angeordnet sein, genauso wie Verzeichnisse und Unterverzeichnisse. Jeder Speicherabschnitt kann weitere, untergeordnete Speicherabschnitte enthalten.

Hinweis

Sie müssen das *dtAutoOpen*-Flag einbinden, wenn Sie *docflags* in der Dokumentregistrierungstabelle registrieren, damit die Verbunddatei I/O korrekt arbeiten kann.

Verbunddateien eignen sich gut dazu, Verbunddokumente zu speichern. Wenn Sie *Open* oder *Commit* aufrufen, erzeugt ObjectComponents automatisch Speicherabschnitte in Ihrer Datei, die jedes Objekt enthalten, die das Dokument enthält. Alle eigentlichen Dokumentdaten werden im Wurzelspeicherabschnitt der Datei gespeichert. Ihre bereits vorhandene Dateidatenstruktur bleibt intakt und wird in einem eigenen Abschnitt gespeichert. Der folgende Code zeigt Ihnen, wie Verbunddokumente gespeichert und geladen werden.

```
// von TOleDocument abgeleitete Dokumentklassendeklaration _DOCVIEWCLASS
TMyDocument : public TOleDocument {
  // Deklarationen

}

// Dokumentklassenimplementierung
bool TMyDocument::Commit(bool force) {
  TOleDocument::Commit(force);      // eingebettete Objekte speichern
                                    // Code, um andere Dokumentdaten zu
                                    // speichern.
}

bool TDrawDocument::Open(int mode, const char far* path) {
  TOleDocument::Open(mode, path);    // alle eingebetteten Objekte laden
                                     // Code, um andere Dokumentdaten zu
                                     // laden
}
```

Das Kommando *TOleDocument::Open* kopiert nicht die Daten aller Objekte in den Speicher. ObjectComponents ist intelligent genug, nur die Daten derjenigen Objekte zu laden, wenn der Benutzer diese aktiviert.

4. Erzeugen des Containers

Sie müssen die richtigen Header einbinden, mit einem unterstützten Speichermodell compilieren und die ObjectComponents- und OLE-Bibliotheken linken, um den Container zu erzeugen.

Einbinden von OLE-Headern

Ein ObjectComponents-Programm benötigt die Klassen, Strukturen, Makros und Symbole, die in den Header-Dateien für die ObjectWindows-OLE-Klassen definiert sind. Die folgende Liste zeigt die Header, die für einen OLE-Container, der das Dok/View-Modell und ein MDI-Rahmenfenster verwendet, benötigt werden.

```
#include <owl/oledoc.h>      // ersetzt DOCVIEW.H
#include <owl/oleview.h>     // ersetzt DOCVIEW.H
#include <owl/olemdifr.h>    // ersetzt MDI.H
```

Eine SDI-Anwendung bindet OLEFRAME.H ein anstelle von OLEMDIFR.H.

Compilieren und Linken

Container, die ObjectComponents und ObjectWindows verwenden, erfordern das Large Memory-Modell. Linken Sie diese mit den OLE- und ObjectComponents-Bibliotheken.

Die Integrierte Entwicklungsumgebung (Integrated Development Environment, IDE) wählt die richtigen Optionen für die Erzeugung, wenn Sie OLE-Unterstützung anfordern. support. Sie müssen eine kurze Make-Datei erzeugen, die die Datei OWLOCFMK.GEN (im Unterverzeichnis EXAMPLES) einbindet, um ein ObjectComponents-Programm von der Kommandozeile aus zu erzeugen. Dies ist beispielsweise die Make-Datei, die das Beispielprogramm *AutoCalc* erzeugt:

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj autocalc.obj
HLP = MYPROGRAM
!include $(BCEXAMPLEDIR)\owlocfmk.gen
```

EXERES und OBJEXE enthalten den Namen der Datei, die erzeugt werden soll, und die Namen der Objektdateien, von denen die Datei erzeugt werden soll.. Die Erweiterung .HLP bezichnet eine optionale Hilfedatei. Schließlich sollte Ihre Make-Datei auch die Datei OWLOCFMK.GEN einbinden.

Nennen Sie Ihre Datei MAKEFILE und geben Sie dies an der Kommandozeilen-Eingabeaufforderung ein:

```
make MODEL=l
```

Make erzeugt unter Verwendung von OWLOCFMK.GEN eine neue Make-Datei, die auf Ihr Projekt zugeschnitten ist. Die neue Make-Datei heißt WIN16L*xx*.MAK. Die beiden letzten Ziffern des Namens geben darüber Auskunft, ob die Make-Datei diagnostische oder Debugging-Versionen der Bibliotheken erzeugt. *01* steht für Debugging-Version, *10* für dia-

gnostische Version, und *11* bedeutet, daß beide Arten von Informationen eingeschlossen sind. Das gleiche Kommando führt die neue Make-Datei aus und erzeugt das Programm. Wenn Sie das Kommando ändern, und MODEL as *d* definieren, ist ide neue Make-Datei WIN16D*xx*.MAK. Diese erzeugt das Programm als dynamische Link-Bibliothek.

Mehr Informationen über die Verwendung von OWLOCFMK.GEN finden Sie in den Instruktionen am Anfang von MAKEFILE.GEN im Verzeichnis EXAMPLES.

Tabelle 20.4 listet die Bibliotheken auf, mit denen ein ObjectComponents-Programm gelinkt werden kann.

Tabelle 20.4: Bibliotheken für die Erzeugung von ObjectComponents-Programmen

Large Model.-Bibliotheken	DLL-Importbibliotheken	Beschreibung
OCFWL.LIB	OCFWI.LIB	ObjectComponents
OWLWL.LIB	OWLWI.LIB	ObjectWindows
BIDSL.LIB	BIDSI.LIB	Klassenbibliotheken
OLE2W16.LIB	OLE2W16.LIB	OLE-System-DLLs
IMPORT.LIB	IMPORT.LIB	Windows-System-DLLs
MATHWL.LIB		Arithmetikunterstützung
CWL.LIB	CRTLDLL.LIB	C-Laufzeitbibliotheken

Die ObjectComponents-Bibliothek muß als erste gelinkt werden, vor der ObjectWindows-Bibliothek. Zu beachten ist auch, daß ObjectComponents RTTI und Exception Handling erfordert. Sie dürfen keine Kommandozeilenoptionen verwenden, diese Features deaktivieren.

Eine ObjectWindows-Anwendung in einen OLE-Container umwandeln

Will man eine ObjectWindows Anwendung in einen OLE-Container umwandeln, erfordert dies einige Veränderungen. Diese Liste gibt eine kurze Beschreibung dieser Änderungen, die dann im darauffolgenden Teil detailliert beschrieben werden.

1. Erstellen der Anwendung.
   - Erstellen Sie ein Anwendungsverzeichnis-Objekt
   - Formen Sie Ihre Anwendungsobjekt um und rufen Sie eine Funktion auf, die es erbt.
2. Registrieren Sie die Anwendung
   - Benutzen sie Registrierungsmakros, um Ihren Container zu beschreiben.
   - Erstellen Sie ein Objekt *TOcRegistrar,* um die Anwendung zu registrieren und zu starten.

3. Erstellen Sie das Client-Fenster.
   - Benutzen Sie ein Client-Fenster bei einer SDI-Anwendung und ein Client-Fenster in Ihrem untergeordneten MDI Fenster, wenn Sie eine MDI-Anwendung haben.
   - Leiten Sie das Rahmenfenster und das Client-Fenster von der neuen ObjectWindows OLE-Klasse ab.
   - Erstellen Sie Rahmen und Client in zwei Schritten, wenn es sich um eine SDI-Anwendung handelt.
   - Für jedes Dokument, das die Anwendung öffnet, erstellen Sie ein Paar von Objektkomponenten.
4. Programmieren sie die Benutzeroberfläche.
   - Wenn Sie Behandlungroutinen für bestimmte Fenstermeldungen übergehen, rufen Sie auf jeden Fall die Behandlungroutine für die Basisklasse auf.
   - Richten Sie Ihre Menüressourcen so ein, daß sie die gemeinsame Nutzung von Menüs unterstützen.
   - Setzen Sie die Standardbefehle für OLE ins Menü *Bearbeiten*.
   - Benutzen Sie Symbolleisten und weisen Sie den vordefinierten Standardbezeichner zu.
5. Erstellen Sie die Anwendung.
   - Binden Sie die neuen ObjectWindows Kopfdateien für OLE.
   - Kompilieren und linken Sie die Anwendung.

Mit dem Befolgen dieser Schritte geben Sie der ObjectWindows-Anwendung folgende Eigenschaften:

Linken	Einbetten
OLE Zwischenablagenoperationen	Ziehen-und-Ablegen Operationen
Direktes Bearbeiten	Symbolleiste und Menüzusammenführung
Verbunde Dokumentspeicherung	OLE 2 Benutzeroberfläche

Im folgenden Teil werden alle Schritte näher erklärt, die Sie benötigen, un eine ObjectWindows-Anwendung in einen OLE 2 Container umzuwandeln. Die Auszüge stammen aus dem Beispiel OWLOCF0.CPP im Verzeichnis EXAMPLES/OWL/TUTORIAL/OLE. Das Beispiel OWLOCF0.CPP stützt sich auf das Beispiel STEP10.CPP aus dem Lernprogramm von ObjectWindows. Es unterstützt kein OLE. OWLOCF1.CPP verändert das erste Programm so, daß es einen OLE-Container unterstützt.

1. Erstellen der Anwendung

Dieser Teil beschreibt die zur Erstellung der Anwendungen benötigten Objektkomponenten. Die Anwendung benötigt ein Anwendungsverzeichnis, und das Objekt, das sie von *TApplication* ableiten, muß ebenfalls von *TOcModule* angeleitet sein.

Definieren Sie ein Objekt für das Anwendungsverzeichnis.

Wird ein DLL von mehr als einer Anwendung oder einem Prozeß benutzt, muß es mehrfache Kopien von globalen, statischen und dynamischen Variablen unterhalten, die seinen momentanen Zustand in jeden Prozeß repräsentieren. Beispielsweise enthält die DLL-Version von ObjectWindows ein Dictionary, das es ihm erlaubt, das Objekt *TApplication* zu ermitteln, das dem momentan aktiven Prozeß entspricht. Wenn Sie Ihre Anwendung in einen DLL-Server verwandeln, muß die Anwendung auch ein Dictionary der Objekte *TApplication* unterhalten, die erstellt werden, wenn sich ein neuer Client an die DLL angliedert. Das Makro DEFINE_APP_DICTIONARY stellt eine einfache und einheitliche Methode zur Erstellung eines Objekts für das Anwendungsverzeichnis zur Verfügung. Fügen Sie die folgende Anweisung in Ihre Deklaration der anderen statischer Variablen ein.

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

Das Makro DEFINE_APP_DICTIONARY definiert die Variable *AppDictionary* korrekt, unabhängig davon, wie die Anwendung aufgebaut ist. In Anwendungen, die die statische Version von ObjectWindows benutzen, erstellt es einfach eine Referenz auf das bereits existierende ObjectWindows Anwendungsverzeichnis. Für DLL-Server hingegen, die die DLL-Version von ObjectWindows benutzen, erklärt das Makro eine Instanz der Klasse *TAppDictionary*. Es ist wichtig, den Namen *AppDictionary* zu benutzen, wenn Sie Ihr Objekt für das Anwendungsverzeichnis erstellen. Damit können Sie die Factory Template-Klassen für die Implementierung einer Factory Callback-Funktion zu benutzen. (Siehe auch Registrieren der Anwendung)

Verändern der Anwendungsklasse

ObjectWindows stellt Anwendungen, die Linken und Einbetten unterstützen, die zugeordnete Klasse *TOcModule* zur Verfügung. Verändern Sie Ihr Anwendungsobjekt so, daß es sich sowohl von *TApplication* als auch *TOcModule* ableitet, wie im folgenden Beispiel gezeigt wird:

```
// Keine OLE-Anwendung
class TScribbleApp : public TApplication {  /* Declarationen */ };

// Neue Deklaration derselben Klasse
class TScribbleApp : public TApplication, public TOcModule { /*
Declarationen */ };
```

Das Objekt *TOcModule* koordiniert die lästigen grundlegenden Arbeiten, die sich auf Registrierung und Speicherverwaltung beziehen. Weiterhin verbindet es Ihre Anwendungsobjekte mit OLE.

Ihre von *TApplication* abgeleitete Klasse muß eine Methode *CreateOleObject* mit der folgenden Signatur zur Verfügung stellen:

```
TUnknown*  CreateOleObject(uint32 options, TDocTemplate* tpl);
```

Diese Methode wird von der Factory Template-Klasse benutzt. Da Container keine OLE-Objekte erstellen, kann ein Container ein *CreateOleObject* erstellen, einfach indem es 0 zurückgibt. Wie im nächsten Kapitel erklärt wird, hat ein Server mehr Arbeit, ein *CreateOleObject* zu implementieren.

```
//
// Keine OLE-Anwendungsklasse
//
class TScribbleApp : public TApplication {
  public:
    TScribbleApp() : TApplication("Scribble Pad") {}

  protected:
    InitMainWindow();

};

//
// Neue Deklaration derselbenKasse
//
class TScribbleApp : public TApplication, public TOcModule {
  public:
    TScribbleApp() : TApplication(::AppReg["description"]){}
    TUnknown*  CreateOleObject(uint32, TDocTemplate*){ return 0; }

  protected:
    InitMainWindow();
```

2. Registrierung eines Containers.

Um eine Anwendung zu registrieren, bauen Sie Registrierungstabellen mit Makros auf. Dann übergeben sie die Tabellen an ein Registrar-Objekt, das die darin enthaltenen Informationen verarbeitet.

Erstellen von Registrierungstabellen.

OLE erfordert, daß Programme sich selbst identifizieren, indem sie einzigartige Namen und Bezeichner registrieren. ObjectWindows stellt Makros zur Verfügung, die es Ihnen erlauben, eine Struktur aufzubauen, die die Registrierungsinformationen enthält. Diese Struktur kann dann benutzt werden bei der Erstellung der Instanz der Anwendung von *TOcRegistrar*. Hier sehen Sie die Befehle, um eine einfache Container-Registrierungsstruktur zu erstellen:

```
REGISTRATION_FORMAT_BUFFER(100)    // Erstellt Puffer für sich
                                   // erweiternde Makros

BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid, "{9B0BBE60-B6BD-101B-B3FF-86C8A0834EDE}")
  REGDATA(description, "Scribble Pad Container")
END_REGISTRATION
```

Das erste Makro, REGISTRATION_FORMAT_BUFFER, legt die Größe eines Puffer fest, den die Makros benötigen, wenn sie zeitweise expandieren. Das Makro REGDATA plaziert Einträge in die Registrierungsstruktur, *AppReg*. Jeder Eintrag in *AppReg* ist eine kleinere Struktur, die ein Schlüsssel wie zum Beispiel *clsid* or *progid* enthält, und einen Wert, der dieser Taste zugewiesen wird. Der zugewiesene Wert ist ein String, der Groß- und Kleinschreibung berücksichtigt. Die Reihenfoge der Tasten innerhalb der Registrierungstabelle ist irrelevant.

Fügen Sie nach der Deklaration des Anwendungs-Verzeichnisses die Registrierungsmakros ein. Da der Wert der *clsid* Taste eine einmalig vorkommende Zahl sein muß, die Ihre Anwendung identifiziert, empfehlen wir die Benutzung des Hilfsprogramms GUIDGEN.EXE. (Im ObjectWindows Referenthandbuch finden Sie unter dem Eintrag *clsid* noch andere Methoden , einen Bezeichner zu generieren.) Natürlich müßen Sie den Wert der Taste *description* entsprechend ändern, um Ihren Container zu beschreiben.

Die Struktur *AppReg*, die in die Beispielcodes eingebaut ist, ist eine Struktur zur Registrierung von Anwendungen. Ein Container kann auch eine oder mehrere *Strukturen zur Registrierung von Dokumenten* aufbauen. Beide Strukturen sind sich ähnlich, aber jede enthält einen anderen Satz von Tasten und Werten. Die Tasten in einer Struktur zur Registrierung von Anwendungen beschreiben Attribute der Anwendung. Eine Struktur zur Registrierung von Dokumenten beschreibt den Dokumententyp, den eine Anwendung erstellen kann. Die Dokumentattribute schließen die Datenformate ein, in denen das Dokument Daten mit der Zwischenablage austauschen kann, seine Dateierweiterung, und den Dokumententypnamen. Die Beispielanwendung OWLOCF1 erstellt keine Strukturen zur Registrierung von Elementen.

Eine Liste aller Registrierungstasten, die ein Container benutzen kann, finden Sie in Tabelle 20.3.

Erstellen eines Registrar-Objekts

Jede ObjectComponents Anwendung muß ein Registrar-Objekt erstellen, um alle Registrierungsaufgaben meistern zu können. Fügen Sie die folgende Zeile hinter der Anweisung #include in Ihre .CCP Hauptdatei ein.

```
static TPointer<TOcRegistrar> Registrar;
```

Die Instanz *TOcRegistrar* wird in Ihrer Funktion *OwlMain* erstellt. Deklariert man den Zeiger als Zeiger vom Typ *TPointer<TOcRegistrar>* anstelle von *TOcRegistrar** stellt man damit sicher, daß die Instanz *TOcRegistrar* gelöscht wird.

Hinweis

Benennen Sie die Variable mit *Registrar*, damit Sie können Sie das Klassentemplate *TOleFactory* nutzen können, um einen Factory Callback zu implementieren.

Im nächsten Schritt müßten Sie Ihre Funktion *OwlMain* abändern, um ein neues Objekt *TOcRegistrar* zuzuweisen, um damit den globalen Zeiger *Registrar* zu initialisieren. Der Konstruktor *TOcRegistrar* erfordert drei Paramenter: die Struktur zur Registrierung von Anwendungen, der Factory Callback der Komponente und den Komandozeilenstring, der die Anwendung aufruft.

Die Registrierungsstruktur erstellen Sie mit den Registrierungsmakros (Siehe Abschnitt »Erstellen von Registrierungstabellen«).

Den Factory Callback erstellen Sie mit einen ObjectWindows Klassentemplate.

Sie können Ihre eigene Callback-Funktion schreiben, wenn Sie möchten, aber Template sind wesentlich einfacher zu benutzen. Für eine ObjectWindows Anwendung, die Linking und Einbettung benutzt, nicht aber Dok/View, heißt die Templateklasse *TOleFactory*. Der Code im Factory Template setzt voraus, daß sie ein Anwendungsverzeichnis erstellt haben, das *AppDictionary* heißt und ein *TOcRegistrar** mit Namen *Registrar* heißt.

Der Kommandozeilen-String kommt von der Methode *GetCmdLine* von *TApplication.*

Hier ist der Code zum Erstellen des Registrars:

```
int OwlMain(int, char*[])
{

  // Erstellung des Registrar-Objekts
  ::Registrar = new TOcRegistrar(::AppReg, TOleFactory<TScribbleApp>(),
                                 TApplication::GetCmdLine());

}
```

Factories werden im ObjectWindows Referenzhandbuch näher erklärt.

Nach der Registrierung des Zeigers *Registrar* muß Ihre Anwendung anstellt von *TApplication::Run TOcRegistrar::Run* aufrufen. Für OLE-Container ruft der *Run* des Registrars einfach den *Run* des Anwendungsobjekts auf, um das Anwendungsfenster zu erstellen und Meldungen zu verar-

beiten. In einem Server jedoch tut *TOcRegistrar::Run* mehr. Wenn Sie die Methode eines Registrar *Run*'s in einem Container benutzen, ist es später wesentlich leichter, die Anwendung entsprechend zu ändern, wenn Sie sie in einen Server umwandeln wollen.

Hier ist der *OwlMain* von OWLOCF1. Aus Gründen der Klarheit sind die üblichen Try-and-Catch Anweisungen weggelassen worden. Die fettgedruckten Zeilen sind der neue Code.

Zuerst:

```
// OwlMain ohne Owl
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  return TScribbleApp().Run();
}
```

Nachdem das Registrar-Objekt eingefügt wurde:

```
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
  ::Registrar = new TOcRegistrar(::AppReg, TOleFactory<TScribbleApp>(),
                                 TApplication::GetCmdLine());
  return ::Registrar->Run();
}
```

3. Erstellung des Client-Fensters

Eine ObjectWindows SDI-Anwendung kann ein Rahmenfenster benutzen, das kein Client-Fenster enhält. Analog kann eine ObjectWindows MDI-Anwendung ein untergeordntes Fenster benutzen, das kein Client-Fenster benutzt. Läßt man das Client-Fenster aus, erschwert dies die Konvertierung der Anwendung von einer Umrahmung zu einer anderen SDI, MDI oder einem dekorativen Rahmen. Es ist auch umständlich beim Aufbau von OLE 2 Anwendungen. Beispielsweise ist es einfacher für das Hauptfenster eines Containers, den Platz für die Werkzeugleiste eines Servers zu schaffen, wenn der Container ein Client-Fenster enthält. Mehr Informationen über die Benutzung von Client-Fenstern erhalten Sie im ObjectWindows Tutorial.

Von einer OLE-Klasse erben

ObjectWindows stellt verschiedene Klassen zur Verfügung, die voreingestelte Implementationen für viele OLE-Operationen enthalten. Um ein bereits existierendes ObjectWindows Programm an OLE anzupassen, verändern Sie seine abgeleiteten Klassen so, daß sie von den OLE-Klassen erben. Eine Liste der OLE-Klassen und den entsprechenden Klassen, die Sie ersetzen, finden Sie in Tabelle 20.1.

Die Klassen *TOleFrame* und *TOleMDIFrame* leiten sich beide von dekorativen Rahmenklassen ab. Die OLE2 Benutzeroberfläche erfordert, daß die Container auf die Handhabung von Werkzeugleisten und Statuszeilen vorbereitet sind. Selbst wenn ein Container keine Dekorationen enthält, kann es sein, daß Server ihre eigenen Dekorationen im Container-fenster anzeigen müßen. Die OLE-Fensterklassen übernehmen diese Verhandlungen für Sie. Der folgende Code zeigt, wie sich die Deklaration für ein Client-Fenster ändert. Die fettgedruckte Schrift markiert die Änderungen.

Zuerst:

```
// Deklaration eines Client-Fensters von OLE
class TScribbleWindow : public TWindow {
  /* declarations */
};
DEFINE_RESPONSE_TABLE1(TScribbleWindow, TWindow);
```

Nachdem die Deklaration so geändert wurde, daß die von einer OLE-fähigen Klasse abgeleitet wird:

```
// Neue Deklaration der selben Fensterklasse
class TScribbleWindow : public TOleWindow {
  /* declarations */
};
DEFINE_RESPONSE_TABLE1(TScribbleWindow, TOleWindow);
```

Verzögerung der Erstellung des Client-Fensters in SDI-Anwendungen

ObjectWindows Anwendungen erstellen ihr Hauptfenster mit der Methode *InitMainWindow* der Klasse, die sich von *TApplication* ableitet. Typischerweise erstellen auch SDI-Anwendungen ihr anfängliches Client-Fenster in der Funktion *InitMainWindow*. Der folgende Code zeigt eine typische Sequenz.

```
void
TDrawApp::InitMainWindow()
{
  // Erstellung des dekorierten Fenster
  TDecoratedFrame* frame = new TDecoratedFrame(0, "Drawing Pad",
                                                new TDrawWindow(0), true);
      // Mehr Deklarationen, um das Hauptfenster zu initialisieren und
      // festzulegen
}
```

Wird diese Sequenz allerdings in OLE-Rahmen- und Client-Klassen benutzt, stellt sie ein Timing-Problem für OLE dar. Das OLE Client-Fenster muß erstellt werde, nachdem der OLE-Rahmen seine Variablen initialisiert hat, die auf ObjectComponents Klassen zeigen. Um diesen Erfordernissen zu genügen, sollte eine SDI OLE-Anwendung nur das Rahmenfenster in der Funktion *InitMainWindow* erstellen. Erstellen Sie das Client-

Fernster mit der Methode *InitInstance* Ihrer Anwendungsklasse. Fettgedruckte Schrift markiert die Änderungen.

```
void
TDrawApp::InitMainWindow()
{
  // Erstellung des Fensterschmuckrahmens
  TOleFrame* frame = new TOleFrame("Drawing Pad", 0, true);

      //
}


void
TDrawApp::InitInstance()
{
  TApplication::InitInstance();

  // Erstellung und Festlegung des Client-Fensters
  GetMainWindow()->SetClientWindow(new TDrawWindow(0));
}
```

Erstellen von ObjectComponents View-und Dokument-Objekten

Für jedes Client-Fenster, das gelinkte oder eingebettete Objekte enthalten kann, müssen Sie ein Objekt *TOcDocument* erstellen, um die eingebetteten OLE-Objekte zu verwalten, und ein Objekt *TOcView* um die Darstellung der OLE-Objekte zu verwalten. Die Methode *CreateOcView* der Klasse *TOleWindow* erstellt sowohl das Container-Dokument als auch das Container-View. Fügen Sie einen Aufruf zu *CreateOcView* in den Konstruktor Ihrer von *TOleWindow* abgeleiteten Klasse ein.

```
// Pre-OLE Deklaration eines Client-Fenster Konstuktors
TScribbleWindow::TScribbleWindow(TWindow* parent, char far* filename)
: TWindow(parent, 0, 0)
{

}


// Neue Deklarationen des Client-Fenster-Konstruktors
TScribbleWindow::TScribbleWindow(TWindow* parent, char far* filename)
: TOleWindow(parent, 0)
{


  // Erstellung eines Objekts OcDocument, das OLE-Teile enthält
  // und eines Objekts OcView, das OLE-Service zur Verfügung stellt.
  CreateOcView(0, false, 0);
}
```

Beachten Sie, daß, anders als beim Konstruktor *TWindow*, der Konstruktor *TOleWindow* keinen Parameter *title* benötigt. Das ist nicht notwendig, weil *TOleWindow* immer der Client eines Rahmens ist. *TWindow* andererseits kann auch als Nicht-Client Fenster benutzt werden – beispielsweise als Pop-Up-Fenster.

Programmierung der Benutzeroberfläche

Die nächsten Änderungen stellen Standardeigenschaften der OLE-Benutzeroberfläche zur Verfügung, wie zum Beispiel Menü-Integration und Ziehen und Ablegen.

Die Behandlung von OLE-abhängigen Meldungen und Ereignissen

ObjectComponents meldet Ihrem Anwendungsfenster OLE-abhängige Ereignisse, indem es die Meldung WM_OCEVENT sendet. Die ObjectWindows-OLE-Klassen stellen für die verschiedenen WM_OCEVENT Ereignismeldungen vorgegebene Handles zur Verfügung. Weiterhin verarbeiten die ObjectWindows-Klassen auch einige der Standard-Fenstermeldungen, und stellen damit der OLE-Benutzeroberfläche zusätzliche Eigenschaften zur Verfügung. Wenn beispielsweise ein Benutzer im Client-Bereich Ihres Container-Fensters doppelklickt, überprüft eine Behandlungsroutine in *TOleWindow* ob dieser Klick über einem eingebetteten Objekt erfolgte, und wenn ja, aktiviert er das Objekt. Ähnlicherweise bewirkt die Methode *TOleWindow::Paint*, daß sich sämtliche eingebetteten Objekte selber zeichnen. Tabelle 20.5 listet die Methoden auf, die von Client-Fenster (*TOleWindow*) und Rahmenfenster (TOleFrame, *TOleMDIFrame*) implementiert werden. Wenn Sie sich über diese Behandlungsroutine in Ihrer abgeleiteten Klasse hinwegsetzen, müssen Sie die Basisklassen-Version aufrufen.

Tabelle 20.5: Standard-Meldungs-Behandlungsroutine, die OLE Funktionen zur Verfügung stellen

Methode	Meldung	Klasse	Beschreibung
EvSize	WM_SIZE	Frame	Meldet den eingebetteten Servern Größenänderungen
EvTimer	WM_TIMER	Frame	Ruft *IdleAction* auf, so daß DLL-Server Befehle aktivieren können
EvActivateApp	WM_ACTIVATEAPP	Frame	Meldet Aktivierungen an eingebettete Server
EvLButtonDown	WM_LBUTTONDOWN	Client	Deaktiviert jedes aktive zum direkten Bearbeiten zuständige Objekt
EvRButtonDown	WM_RBUTTONDOWN	Client	Zeigt ein Pop-Up Menü, wenn der Mauszeiger auf einem eingebetteten Objekt steht
EvLButtonDblClk	WM_LBUTTONDBLCLK	Client	Aktiviert jedes eingebettete Objekt unter dem Mauszeiger
EvMouseMove	WM_MOUSEMOVE	Client	Erlaubt dem Benutzer, ein eingebettetes Objekt zu bewegen oder dessen Größe zu verändern

EvLButtonUp	WM_LBUTTONUP	Client	Informiert das ausgewählte Objekt über Größen- oder Positionsänderungen
EvSize	WM_SIZE	Client	Informiert das Objekt *TOcView*, daß das Fenster seine Größe geändert hat
EvMdiActivate	WM_MDIACTIVATE	Client	Informiert das Objekt *TOcView*, daß das Fenster seine Größe geändert hat
EvMouseActivate	WM_MOUSEACTIVATE	Client	Gibt die Meldung weiter an das höchste übergeordente Fenster und gibt den Code zurück, mit dem das Client-Fenster aktiviert wird
EvSetFocus	WM_SETFOCUS	Client	Meldet jede Änderung im Fokus dem für direktes Bearbeiten zuständigen Servers
EvSetCursor	WM_SETCURSOR	Client	Ändert die Form des Mauszeigers über einem eingebetteten Objekt
EvDropFiles	WM_DROPFILES	Client	Bettet fallengelassene Dateien ein
Paint	WM_PAINT	Client	Bewirkt, daß eingebettete Objekte zeichnen
EvCommand	WM_COMMAND	Client	Verarbeitet Befehls-ID's von Aktionsarten
EvCommandEnable	WM_COMMANDENABLE	Client	Verarbeitet Befehls-ID's von Aktionsarten.

In einigen Fällen werden Sie wissen müßen, welche Aktion die Behandlungsroutine der Basisklasse ausführte, bevor Sie entscheiden, was Ihre überschriebene Behandlungsroutine tun soll. Das trifft besonders auf mausabhängige Meldungen zu. Übernimmt beispielsweise die Basisklasse eine Doppelklick-Aktion, und der Benutzer möchte damit ein Objekt aktivieren, wollen Sie vermutlich nicht, daß Ihr Code den Doppelklick als einen anderen Befehl interpretiert. Der folgende Code zeigt Ihnen, wie man eine Aktion mit einer Basisklassen-Behandlungsroutine koordiniert. Diese drei Prozeduren lassen den Benutzer mit der Maus auf der Oberfläche des Client-Fensters zeichen.

```
void
TMyClient::EvLButtonDown(uint modKeys, TPoint& pt)
{
  if (!Drawing) {
    SetCapture()
    Drawing = true;
        // zusätzliche GDI-Aufrufe zum Zeichen auf dem Bildschirm
  }
}

void
TMyClient::EvMouseMove(uint modKeys, TPoint& pt)
{
  if (Drawing) {
        // zusätzliche GDI-Aufrufe zum Zeichen auf dem Bildschirm
  }
```

```
}

void
TMyClient::EvLButtonUp(uint modKeys, TPoint& pt)
{
  if (Drawing) {
    Drawing = false;
    ReleaseCapture();
  }
}
```

Als OLE-Container kann das Client-Fenster allerdings eingebettete Objekte enthalten. Mausaktionen, die über diesen Objekten ausgeführt werden, sollten also keine Zeichenoperationen durchführen. Dieser Code zeigt, wie die Behandlungsroutinen aktualisiert und geprüft werden für OLE-bezogene Verarbeitung. Fettdruck verdeutlicht die Veränderungen.

```
void
TMyClient::EvLButtonDown(uint modKeys, TPoint& pt)
{
  TOleWindow::EvLButtonDown(modKeys, pt);

  if (!Drawing && !SelectEmbedded()) {
    SetCapture()
    Drawing = true;
        // zusätzliche GDI-Aufrufe zum Zeichen auf dem Bildschirm
  }
}

void
TMyClient::EvMouseMove(uint modKeys, TPoint& pt)
{
  TOleWindow::EvMouseMove(modKeys, pt);

  if (Drawing && !SelectEmbedded()) {
        // zusätzliche GDI-Aufrufe zum Zeichen auf dem Bildschirm
  }
}

void
TMyClient::EvLButtonUp(uint modKeys, TPoint& pt)
{
  if (Drawing && !SelectEmbedded()) {
    Drawing = false;
    ReleaseCapture();
  }

  TOleWindow::EvLButtonUp(modKeys, pt);
}
```

Die Methode *SelectEmbedded* wird von *TOleWindow* vererbt. Sie gibt true zurück, wenn ein eingebettetes Objekt momentan bewegt wird. Das Client-Fenster ruft sie auf, um zu entscheiden, ob eine Mausmeldung bereits von der OLE-Basisklasse verarbeitet wurde.

Typischerweise muß Ihre abgeleitete Klasse die Basisklassen-Behandlungsroutinen aufrufen, bevor sie irgendwelche Ereignisse oder Meldungen verarbeiten kann. Die Behandlungsroutine *EvLButtonUp* allerdings ruft die Basisklasse erst zum Schluß auf. Dies erlaubt der Behandlungsroutine, sich auf *SelectEmbedded* zu verlassen, welches wahrscheinlich zurückgesetzt wird, nachdem *TOleWindow* die Meldung "*Maus nach Oben*" verarbeitet hat.

Unterstützung von ;Zusammenführung vonMenüzusammenführung

Die Menüleiste eines OLE-Containers mit einem aktiven Objekt besteht aus einzelnen Teilen der normalen Menüs von Container und Server. Der Container stellt Pop-Up-Menüs zur Verfügung, die den Anwendungsrahmen oder Dokumente behandeln. Der Server andererseits stellt das Menü *Bearbeiten* zur Verfügung, das Hilfe-Menü und alle anderen Menüs, die der Benutzer zur Manipulation der aktivierten Objekte benutzen kann.

OLE teilt das Hauptmenü einer Menüleiste in sechs Gruppen ein. Jede Gruppe ist eine Reihe aneinandergrenzender Dropdown-Menüs und besteht aus keinem oder mehreren Pop-Up-Menüs. Die Menügruppen heißen Datei, Bearbeiten, Container, Objekt, Fenster und Hilfe. Diese Gruppennamen sind lediglich eine Konvention. Sie legen eine gemeinsame Organisation verwandter Befehle nahe, aber Sie können die Befehle gruppieren, wie immer Sie es möchten.

Wenn ein Container oder ein Server alleine operiert, stellt er die Menüs für alle sechs Gruppen zur Verfügung. Während einer Editiersitzung mit direktem Bearbeiten allerdings ist der Container für die Kontrolle der Gruppen Datei, Container und Fenster zuständig, während der Server die Gruppen Bearbeiten, Objekt und Hilfe übernimmt.

Die Klasse *TMenuDescr* übernimmt automatisch alle Verhandlungen über Ihren Menüressourcen zwischen Server und Container. Sie bezeichnen einfach die verschiedenen Menügruppen innerhalb Ihrer Menüressource, und ObjectWindows zeigt die richtigen Menüs zur richtigen Zeit an.

Um anzuzeigen, wo innerhalb Ihrer Menüressource eine Gruppe beginnt und endet, fügen Sie das Menüelement SEPARATOR zwischen Ihnen ein. Sie müssen alle sechs Gruppen markieren, selbst wenn einige davon leer sein sollten. Die Klasse *TMenuDescr* sucht nach diesen Trennzeichen, wenn sie ein Menü von einer Ressource lädt. Sie entfernt die Trennzeichen zwischen den obersten Einträgen und baut eine Struktur auf, die die Anzahl der Pop-Up-Menüs, die jeder Gruppe zugewiesen werden, speichert. Diese Information erlaubt es ObjectWindows, die Server-Menüs mit den Container-Menüs in der Menüleiste des Containers zusammenzuführen.

Das folgende Menüressourcen-Script illustriert, wie ein einfaches Anwendungsmenü definiert wird, bevor es in Gruppen unterteilt wird. Es stammt aus STEP10.RC aus dem ObjectWindows Tutorial.

```
COMMANDS MENU
{
  pop-up "&File"
  {
    MENUITEM "&New",      CM_FILENEW
    MENUITEM "&Open",     CM_FILEOPEN
    MENUITEM "&Save",     CM_FILESAVE
    MENUITEM "Save &As", CM_FILESAVEAS
  }

  pop-up "&Tools"
  {
    MENUITEM "Pen &Size",  CM_PENSIZE
    MENUITEM "Pen &Color", CM_PENCOLOR
  }

  pop-up "&Help"
  {
    MENUITEM "&About",     CM_ABOUT
  }
}
```

Der Menüeintrag *File* gehört zur OLE-Menügruppe *File*. Das Menü *Tools* erlaubt es dem Benutzer, das Anwendungsdokument zu bearbeiten, also gehört es zur Gruppe Bearbeiten. Diese Anwendung enthält keine Menüs, die zu der Objekt-, Container- oder Fenstergruppe gehören. Und zuletzt gehört das Menü Hilfe zur Gruppe Hilfe.

Der folgende Code ist eine modifizierte Version des selben Menüs, mit eingefügten SEPARATOR Unterteilern, die anzeigen, wo eine Gruppe endet und die nächste beginnt. Fettdruck verdeutlicht die Veränderungen.

```
COMMANDS MENU
{
  pop-up "&File"
  {
    MENUITEM "&New",      CM_FILENEW
    MENUITEM "&Open",     CM_FILEOPEN
    MENUITEM "&Save",     CM_FILESAVE
    MENUITEM "Save &As", CM_FILESAVEAS
  }

  MENUITEM SEPARATOR          // Ende der Gruppe Datei, Beginn der Gruppe
                              // Bearbeiten

  pop-up "&Tools"
```

```
  {
    MENUITEM "Pen &Size",  CM_PENSIZE
    MENUITEM "Pen &Color", CM_PENCOLOR
  }

  MENUITEM SEPARATOR          // Ende der Gruppe Bearbeiten, Beginn der
                              // Gruppe Container
  MENUITEM SEPARATOR          // Ende der Gruppe Container, Beginn der
                              // Gruppe Onjekt
  MENUITEM SEPARATOR          // Ende der Gruppe Objekt, Beginn der Gruppe
                              // Fenster
  MENUITEM SEPARATOR          // Ende der Gruppe Fenster, Beginn der
                              // Gruppe Hilfe

  pop-up "&Help"
  {
    MENUITEM "&About",      CM_ABOUT
  }
}
```

Fügen Sie Unterteiler in Ihre Anwendungsmenüs ein, um die verschiedenen Menügruppen zu kennzeichnen. Verändern Sie dann Ihren Code mit Hilfe der Methode *SetMenuDescr,* wenn Sie das Rahmenfenster-Menü zuweisen. Dieses Beispiel zeigt eine Menüzuweisung vor und nach der Menüzusammenführung. Fettdruck verdeutlicht die Veränderungen.

Zuerst:

```
// Ursprüngliche Menüzuweisung
void
TScribbleApp::InitMainWindow()
{
  TDecoratedFrame* frame;
      // Initialize frame and decorations etc. etc.

  // Rahmenmenüzuweisung
  frame->AssignMenu("COMMANDS");
}
```

Nach Hinzufügen der Gruppenindikatoren in den Menüs:

```
void
TScribbleApp::InitMainWindow()
{
  TOleFrame* frame;
      // Initialisiere Rahmen, Verzierungen etc.

  // Weise das Rahmenmenü zu
  frame->SetMenuDescr(TMenuDescr("COMMANDS"));
}
```

Statt Unterteiler zu benutzen, um anzuzeigen, welcheDrop-Down-Menüs zu jeder Gruppe gehören, können Sie auch den Konstruktor *TMenuDescr* benutzen, dessen Parameter eine Zählung für jede Gruppe akzeptieren. Die Konstruktoren *TMenuDescr* sind genauer im ObjectWindows Referenzhandbuch beschrieben.

Aktualisieren des Menüs Bearbeiten

Ein OLE-Container setzt seine OLE-Befehle ins Menü *Bearbeiten*. Tabelle 20.2 listet alle diese Befehle auf. Die Klasse *TOleWindow* enthält die vorgegebenen Implementationen für diese Befehle. Sie ruft bei Bedarf die Standarddialogfelder auf und verarbeitet die Antworten des Benutzers. Alles, was Sie tun müssen, ist den Befehl in das Menü *Bearbeiten* Ihres Rahmenfensters aufzunehmen. Es ist nicht nötig, alle sechs Befehle zu unterstützen, aber jeder Container sollte zumindes ein *Einfügen*-Objekt unterstützen, damit der Benutzer neue Objekte zu dem aktuellen Dokument hinzufügen kann; sowie das Objekt Bearbeiten, damit der Benutzer das aktuelle Objekt aktivieren kann.

ObjectWindows definiert in *owl/oleview.rh* Standardbezeichner für die Befehle des OLE-Menüs *Bearbeiten*. Aktualisieren Sie Ihre Ressourcendatei, so daß sie die Header-Datei enthält, und benutzen Sie die Standardbezeichner, um OLE-Befehle in das Menü Bearbeiten einzufügen.

```
#include <owl/oleview.rh>
#include <owl/edit.rh>

COMMANDS MENU
{
      // Hier ist das Menü Datei

  MENUITEM SEPARATOR
  pop-up "&Edit"
  {
    MENUITEM "&Undo\aCtrl+Z",            CM_EDITUNDO
    MENUITEM Separator
    MENUITEM "&Cut\aCtrl+X",             CM_EDITCUT
    MENUITEM "C&opy\aCtrl+C",            CM_EDITCOPY
    MENUITEM "&Paste\aCtrl+V",           CM_EDITPASTE
    MENUITEM "Paste &Special...",   CM_EDITPASTESPECIAL
    MENUITEM "Paste &Link",          CM_EDITPASTELINK
    MENUITEM "&Delete\aDel",             CM_EDITDELETE
    MENUITEM SEPARATOR
    MENUITEM  "&Insert Object...",  CM_EDITINSERTOBJECT
    MENUITEM  "&Links...",           CM_EDITLINKS
    MENUITEM  "&Object",             CM_EDITOBJECT
    MENUITEM SEPARATOR
    MENUITEM  "&Show Objects",     CM_EDITSHOWOBJECTS
  }
```

```
        // Hier sind die anderen Menüs
}
```

Eine Werkzeugleisten-ID zuweisen

Wenn Ihr OLE-Container eine Werkzeugleiste enthält, weisen Sie ihm den vordefinierten Bezeichner IDW_TOOLBAR zu. ObjectWindows muß instande sein, die Werkzeugleiste zu finden, wenn ein Server seine eigene Werkzeugleiste im Container-Fenster darstellen muß. Wenn ObjectWindows die alte Werkzeugleiste identifizieren kann, ersetzt es sie zeitweise mit der neuen des Servers. Damit ObjectWindows die Werkzeugleiste des Containers identifizieren kann, muß der Container IDW_TOOLBAR als Fenster-ID benuten.

```
TControlBar* cb = new TControlBar(parent);
cb->Attr.Id = IDW_TOOLBAR;
```

Die Methode *TOleFrame::EvAppBorderSpaceSet* benutzt IDW_TOOLBAR als voreingestellte Implementierung. Ein Container kann seine eigene Implementation zur Verfügung stellen, um komplexere Situationen zu handhaben, wie zum Beispiel die Zusammenführung mehrerer Werkzeugleisten.

5. Einen Container aufbauen

Ein Container muß einenOLE ObjectWindows-Header einschließen, mit einem unterstützten Speichermodell compiliert werden, und mit den richtigen Bibliotheken verbunden sein.

OLE Header einschließen

ObjectWindows stellt OLE-bezogenen Klassen, Strukturen, Makros und Symbole in verscheidenen Header-Dateien zur Verfügung. Die folgende Liste zeigt die Header, die für einen OLE-Container gebraucht werden, wenn ein SDI-Rahmenfenster benutzt wird.

```
#include <owl/oleframe.h>
#include <owl/olewindo.h>
#include <ocf/ocstorag.h>
Eine MDI-Anwendung würde olemdifr.h statt oleframe.h einschließen.
```

Kompilieren und Verbinden

ObjectWindows Container und Server müssen mit dem Speichermodell **Large** kompiliert werden. Sie müssen mit den OLE, den ObjectComponents und den ObjectWindows Bibliotheken gelinkt werden.

Umwandeln eines C++ Programms in einen OLE-Container

Beim Erstellen jedes neuen Programms sollten Sie die Verwendung von ObjectWindows ins Auge fassen, um sich die Arbeit zu erleichtern. Die ObjectWindows-Bibliothek enthält Bausteine, die einige Aufgaben erfüllen, die in allen ObjectComponents-Programmen anfallen. Programme,

in denen ObjectWindows nicht genutzt wird, müssen diese Routinearbeiten selbst realisieren.

Auch die Benutzung des AppExpert und ObjectWindows erspart Ihnen einige Arbeit. ObjectComponents funktioniert auch in "einfachen" C++ Programmen ohne ObjectWindows.

Die folgende Liste umreißt kurz die notwendigen Änderungen zur Umwandlung einer C++ Anwendung in einen ObjectComponents-Container. Die folgenden Abschnitte erläutern die einzelnen Schritte dann detaillierter.

1. Registrieren sie die Anwendung.
   - Erstellen sie eine Registrierungstabelle mit den entsprechenden Registriermakros.
   - Erstellen sie ein Registrierungsobjekt und starten dessen *CreateOcApp* Funktion.
   - Erstellen sie ein *TOleAllocator* Object, um die OLE Bibliotheken zu initialisieren.
2. Erstellen sie ein Sichtfenster, in dem ein geöffnetes Dokument anzeigt werden kann.
   - Erstellen sie das Sichtfenster, modfizieren dessen Größe und entfernen es zusammen mit dem Hauptfenster.
   - Erstellen sie zwei Hilfsobjekte, *TOcDocument* und *TOcView*, um die OLE-Aspekte eines zusammengesetzten Dokuments zu bearbeiten.
   - Lassen sie die WM_OCEVENT Botschaft durch das Sichtfenster bearbeiten.
   - Schreiben sie Behandlungsroutinen, die durch ausgewählte Ereignisse von ObjektComponents ausgelöst werden.
   - Geben sie das zusammengesetzte Dokument im Sichtfenster aus.
   - Aktivieren und deaktivieren sie Objekte in Übereinstimmung mit Anwenderaktionen.
3. Programmieren sie das Hauptfenster so, daß dieses OLE-Kommandos und -Ereignisse verarbeitet.
   - Übergeben Sie ein Handle des Hauptfensters an das *TOcApp*-Objekt.
   - Das Hauptfenster soll die WM_OCEVENT Botschaft verarbeiten.
   - Schreiben sie Behandlungsroutinen für anwendungsspezifische Ereignisse markierter Objektkomponenten.
   - Erweitern sie das Menü *Bearbeiten* um OLE Befehle (wie Einfügen eines Objekt) und schreiben Behandlungsroutinen für diese Zusätze.
4. Erzeugen Sie das Programm.

- Nehmen sie die Header der Objektkomponenten in Ihr Programm auf.
- Übersetzen Sie das Programm im Speichermodell **Large**. Linken sie unter Einbeziehung der Bibliotheken für OLE- und ObjectComponents.

Die folgenden Anschnitte illustrieren jeden Schritt anhand von Beispielprogrammen des Verzeichnisses EXAMPLES/OCF/CPPOCF. Die Quelltextdateien namens CPPOCF0 realisieren eine Windows-Anwendung ohne OLE. CPPOCF1 erweitert das ersteProgramm zu einem OLE-Container. Die Programmbeispiele dieser Erläuterungen finden Sie in CPPOCF1. Das selbe Verzeichnis enthält auch CPPOCF2, eine OLE Server-Anwendung. Kapitel 21 illustriert mittels CPPOCF2 die Erstellung eines OLE-Servers ohne ObjectWindows.

CPPOCF1 ist eine einfache Anwendung, die fundamentale Funktionen eines Containers unterstützt: Registrieren der Anwendung, Erstellen von Objekten zur Initialisierung neuer Dokumente und Einbinden eines Objekts in ein Dokument. Im Programmtext der OLE-Klassen von ObjectWindows, wie *TOleWindow* und *TOleView*, erhalten Sie weitere Anregungen zur Implementierung anderer Eigenschaften.

Die folgenden Erläuterungen erklären im wesentlichen nur die Unterschiede der Programmtexte von CPPOCF0 zu CPPOCF1, die spezifisch für ObjectComponents und OLE sind, wie der Aufruf von *RegisterClass* für ein neues untergeordnetes Fenster.

1. Registrieren eines Containers

In drei Schritten geben Sie dem System notwendige Informationen über Ihren Container: Aufbau einer Registrierungstabelle, Tabellenübergabe an ein Registrar-Objekt, und Erstellen eines Objekts zur Speicherreservierung.

Aufbau einer Registrierungsungstabelle

Ein Container benutzt die Registrierungsmakros, um eine Registrierungstabelle aufzubauen, die die Anwendung beschreibt.. Ein Container muß nur dann eine Registrierungstabelle für Dokumente erstellen, wenn er selbst als Quelle für Verknüpfungen genutzt werden kann. (Ein Container ist eine Quelle für Verknüpfungen, wenn er anderen Containern Verknüpfungen zu Objekten seiner Dokumente ermöglicht.)

Es folgt eine Registrierungstabelle aus CPPOCF1:

```
REGISTRATION_FORMAT_BUFFER(100)
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,        "{8646DB80-94E5-101B-B01F-00608CC04F66}")
  REGDATA(progid,       APPSTRING ".Application.1")
  REGDATA(description,  "Sample container")
END_REGISTRATION
```

Die Header-Datei der Anwendung enthält folgende Zeile:

```
#define APPSTRING   "CppOcf1"
```

Der String *progid* lautet also "CppOcf1.Application.1."

Die Registrierungsmakros erstellen eine Struktur des Typs *TRegList*. Jeder Eintrag dieser Struktur enthält einen Schlüssel, wie *clsid* oder *progid*, und einen diesem Schlüssel zugeordneten Wert. Intern finden Objektkomponenten Werte, indem sie nach den zugehörigen Schlüsseln suchen. Die Reihenfolge der Schlüssel spielt keine Rolle. Weitere Informationen über Felder, die ein Container registrieren kann finden Sie in Tabelle 20.3. Über Registrierung in ObjectComponents erfahren Sie mehr im Abschnitt »Registrierung verstehen«.

Erzeugen des Registrar-Objekts

Das Registrar-Objekt zeichnet Anwendungsinformationen in der System-Registrierungsdatenbank auf, verarbeitet die OLE-Schalter in der Kommandozeile der Anwendung und benachrichtigt OLE, daß der Server gestartet wurde. CPPOCF2 deklariert einen statischen Zeiger für das Registrar-Objekt:

```
TOcRegistrar* OcRegistrar = 0;
TOcApp*       OcApp       = 0;
```

Die zweite Variable, *OcApp*, zeigt auf das Verbindungsobjekt, das die OLE-Schnittstellen bereitstellt, über die die Anwendung mit OLE kommuniziert. Der Registrar erzeugt das Objekt *TOcApp* in *WinMain*.

Erzeugen Sie den Registrar bei der Initialisierung Ihrer Anwendung in *WinMain*. Rufen Sie die Methode *Run* des Registrars auf, statt in einer Schleife auf Botschaften zur Verarbeitung zu warten. Terminiert *Run*, so kann die Anwendung ebenfalls beendet werden. Geben Sie den Registrar frei, bevor die Anwendung terminiert. Folgender Ausschnitt der CPPOCF1 *WinMain* Funktion faßt nochmals alle Schritte zusammen.

```
int PASCAL
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  char far* lpCmdLine, int nCmdShow)
{
  try {
    TOleAllocator allocator(0); //für OLE2 notwendig
    MSG msg;

    // Initialisierung der OCF Objekte
    OcRegistrar = new TOcRegistrar(::AppReg, 0,
                                   string(lpCmdLine), 0);
    OcRegistrar->CreateOcApp(OcRegistrar->GetOptions(), OcApp);

          // Initialisierung einzelner Instanzen und Tasks möglich
```

```
    // Standard Windows Schleife zur Verarbeitung von Botschaften
    while (GetMessage(&msg, 0, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  catch (TXBase& xbase) {

MessageBox(GetFocus(), xbase.why().c_str(), "Exception caught", MB_OK);
  }

  // Freigabe des Registrar-Objekts
  delete OcRegistrar;
  return 0;
}
```

Vier Parameter werden an den Konstruktor von *TOcRegistrar* übergeben:

- *::AppReg*, ist die bereits mit Registrierungsmakros erzeugte Registrierungsstruktur für Anwendungen. (Siehe Abschnitt »Erzeugung einer Registrierungstabelle«)
- *ComponentFactory* ist eine CallBack-Funktion, die im nächsten Abschnitt beschrieben wird.
  Sie ist für die bedarfsgerechte Erzeugung aller OLE-Komponenten einer Anwendung, einschließlich der Anwendung selbst, zuständig. In CPPOCF1 ist auch die Warteschleife zur Verarbeitung eintreffender Botschaften in der Callback-Funktion.
- *cmdLine* ist ein *string*-Objekt, das die Kommandozeile der Anwendung enthält.
  Der Registrar durchsucht *cmdLine* nach OLE-Schaltern, wie Automation oder Embedding, und setzt dementsprechend interne Laufzeitschalter.
- 0 ist ein Null-Zeiger auf eine Liste von Dokumenten.
  Da CPPOCF1 keine Dokumenttypen registriert, ist diese Liste leer. Ein Container registriert Dokumenttypen, wenn er als Verknüpfungsquelle genutzt werden soll.

Neben dem Aufzeichnen von Informationen in der Registrierungsdatenbank, erzeugt das Registrar-Objekt das *TOcApp*-Verbindungsobjekt, sobald *CreateOcApp* aufgerufen wird.

Erzeugung eines Hauptspeicherverwalters

Die *WinMain* Prozedur erzeugt zu Beginn einen *TOleAllocator*:

```
TOleAllocator allocator(0);      // Standardspeicherverwaltung nutzen
```

Der Konstruktor des Speicherverwalters initialisiert die OLE-Bibliotheken und der Destruktor gibt diese frei, wenn der Gültigkeitsbereich eines Objekts verlassen wird. Die Übergabe von 0 an den Konstruktor bedeu-

tet, daß zum Standard von OLE gehörenden Hauptspeicherverwaltungsfunktionen nutzen soll, wenn die Anwendung Hauptspeicher anfordert.

2. Erzeugen eines Sichtfensters

Objektkomponenten legen bezüglich des Programmdesigns folgendes fest: ein zusammengesetztes Dokument muß ein eigenes vom Hauptfenster unabhängiges Fenster haben. Um den Unterschied deutlich zu machen, nennt man das Hauptfenster das Rahmenfenster, weil es einen sichtbaren Rahmen des Stils WS_THICKFRAME hat. Das zweite Fenster hat keinen sichtbaren Rahmen. Dieses nennt man Sichtfenster, weil dort die Anwendung ihre Daten anzeigt. Das Sichtfenster füllt den Client-Bereich des Rahmenfensters völlig aus, so daß der Anwender scheinbar nur ein Fenster vor sich hat. Objektkomponenten benötigen das Sichtfenster aber, weil ein Teil der Botschaften an die Anwendung, ein anderer an die Ansicht gerichtet ist.

In einer SDI-Anwendung wie dem Beispiel CPPOCF1, steuert das Rahmenfenster das Sichtfenster. Empfängt das Rahmenfenster eine Botschaft WM_SIZE, so verschiebt es die Ansicht, so daß diese weiterhin im Client-Bereich liegt und diesen ausfüllt. Erhält es eine WM_CLOSE Botschaft, so schließt es sich selbst und das Sichtfenster.

In einer MDI Anwendung erzeugt jedes untergeordnete Fenster seine eigene Ansicht. Das untergeordnete MDI-Fenster verhält sich wie ein SDI-Rahmenfenster: es erzeugt und steuert eine Ansicht für das Dokument, das es darstellt.

Erzeugen, Größe ändern und Löschen des Sichtfensters

Vor der Erzeugung des Sichtfenster muß die Anwendung eine Klasse für dieses registrieren. CPPOCF1 registriert beide Klassen in *InitApplication*.

CPPOCF1 erzeugt das Sichtfenster im eigenen Factory, weil dieses die Anforderungen für neue Dokumente bearbeitet. Der Programmtext für das Sichtfenster verbindet, wie Sie gleich sehen werden, das neue Dokument mit OLE, indem es ein paar Hilfsobjekte von ObjectComponents erstellt. Das factory benutzt folgende Funktion um das Sichtfenster zu erzeugen:

```
HWND CreateViewWindow(HWND hwndParent)
{
  HWND hwnd = CreateWindow(VIEWCLASSNAME, "",
    WS_CHILD | WS_CLIPCHILDREN | WS_CLIPSIBLINGS | WS_VISIBLE | WS_BORDER,
    10, 10, 300, 300,
    hwndParent, (HMENU)1, HInstance, 0);
  return hwnd;
}
```

CPPOCF1 verändert die Größe und löscht das Sichtfenster, wenn das Rahmenfenster Botschaften der Art WM_SIZE beziehungsweise WM_CLOSE empfängt.

```
void
MainWnd_OnSize(HWND hwnd, UINT /*state*/, int /*cx*/, int /*cy*/)
{
  if (IsWindow(HwndView)) {
    TRect rect;
    GetClientRect(hwnd, &rect);
    MoveWindow(HwndView, rect.left, rect.top, rect.right, rect.bottom,
true);
  }
}

void
MainWnd_OnClose(HWND hwnd)
{
  if (IsWindow(HwndView))
    DestroyWindow(HwndView);
  DestroyWindow(hwnd);
}
```

Das Sichtfenster füllt den Client-Bereich des Rahmenfensters immer völlig. Öffnet und schließt der Anwender Dokumente oder bindet er Objekte ein, werden alle Veränderungen im Sichtfenster angezeigt.

Erzeugen von TOcDocument und TOcView

Bindet der Anwender mehrere Objekte in das Sichtfenster des Containers ein, so werden diese Teil eines zusammengesetzten Dokuments. Unterstützt der Container Ein- und Ausgabe von Dateien, so kann der Anwender Dokumente auch speichern und laden.

Für jedes Dokument, das der Container öffnet, benötigt er ein Sichtfenster und zwei Hilfsobjekte: *TOcDocument* und *TOcView*. Das Hilfsobjekt des Dokuments verwaltet eine Menge von in das Dokument eingefügten Objekten. Das Hilfsobjekt der Ansicht verbindet das Dokument mit OLE. Genauer gesagt, stellt es Schnittstellenfunktionen bereit, die OLE zur Kommunikation mit dem Dokument aufrufen kann. Teilt OLE dem Darstellungsobjekt mit, daß ein relevantes Ereignis stattgefunden hat, so sendet das Darstellungsobjekt eine Botschaft an das Sichtfenster. (Die beiden nächsten Abschnitte zeigen, wie diese Botschaften zu verarbeiten sind.)

Das Container-Beispiel CPPOCF1 deklariert zwei globale Zeiger, die zwei Hilfsobjekte aufnehmen.. (Jedes Programm, das mehr als ein Dokument gleichzeitig offen hält, benötigt mehr als ein Variablenpaar.)

```
TOcDocument*  OcDoc      = 0;
TOcView*      OcView     = 0;
```

CPPOCF1 erzeugt und entfernt die beiden Hilfsobjekt beim Erstellen und Löschen der Ansicht, die das Dokument darstellt.

```
bool
ViewWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  OcDoc  = new TOcDocument(*OcApp);    // Erzeugen der Dokumentenhilfe
  OcView = new TOcView(*OcDoc);        // Erzeugen des View-Konnektors
  if (OcView)
    OcView->SetupWindow(hwnd);         // Verbinden der Ansicht mit dem
Fenster
  return true;
}

void
ViewWnd_OnDestroy(HWND /*hwnd*/)
{
    // Hier ist die Deaktivierung von Objekten möglich (Erklärung folgt).

  if (OcView)
    OcView->ReleaseObject();      // Ansicht nicht löschen, da COM-Objekt
  OcDoc->Close();                 // Freigabe des Servers für alle
eingebundenen Objekte.
  delete OcDoc;                   // Löschen der Dokementenhilfe; kein
COM-Objekt
}
```

Die Verarbeitungsroutine der Ansicht für Botschaften des Typs WM_CREATE erzeugt beide Hilfen und ruft danach *OcView->SetupWindow* auf. Die Methode *SetupWindow* teilt dem *TOcView*-Objekt mit, wohin Botschaften über Ereignisse zu senden sind. In diesem Fall sendet es Botschaften zum Sichtfenster *hwnd*. Das Sichtfenster empfängt daraufhin Botschaften des Typs WM_OCEVENT.

Wird das Sichtfenster gelöscht, so tätigt es drei Aufrufe, um die Hilfsobjekte zu löschen. *OcView->ReleaseObject* signalisiert, daß das Sichtfenster die Bearbeitung des *TOcView*-Konnektorobjekts abgeschlossen hat. Den Aufruf von delete für ein *TOcView*-Objekt sollte man vermeiden, weil das OLE System noch immer Informationen benötigen könnte, bevor die Ansicht beendet werden kann. *ReleaseObject* teilt dem *TOcView*-Objekt mit, daß es nicht mehr benötigt wird. Die View terminiert sich daraufhin selbst, sobald kein anderer OLE-Client sie mehr nutzt. Der *TOcView* Destruktor ist **protected**, um zu verhindern, daß er direkt aufgerufen wird.

(Ein Container-Objekt kann nur OLE-Clienten haben, wenn es Unterstützung für seine Rolle als Verbindungsquelle registriert. In diesem Falle können andere Anwendungen Verbindungen zu Objekten im Dokument des Containers erzeugen.)

Andererseits ist das *TOcDocument*-Objekt kein Verbindungsobjekt und kann also wie üblich mit **delete** entfernt werden. Zuerst sollten Sie jedoch *Close* aufrufen, um Server-Anwendungen freizugeben, die OLE angestossen hat, um die Funktionalität für jedes verbundene oder eingebettete Objekt bereitzustellen.

Verarbeiten von WM_OCEVENT

Da die Methode *TOcView::SetupWindow* den *TOcView*-Konnektor an das Sichtfenster gebunden hat, sendet der Konnektor seine Ereignisbotschaften an das Fenster. Alle ObjectComponents-Ereignisse werden in der Botschaft WM_OCEVENT versendet, so daß die Prozedur des Sichtfensters auf WM_OCEVENT antworten muß.

```
long CALLBACK _export
ViewWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
   // hier können weitere Routinen zur Verarbeitung von Botschaften stehen
    HANDLE_MSG(hwnd, WM_OCEVENT,  ViewWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}
```

Das Makro HANDLE_MSG zur Verarbeitung der Botschaft WM_OCEVENT ist im ocf/ ocfevx.h header definiert. Derselbe Header definiert eine weiteren Routine zur Reaktion auf Botschaften, die in der Botschaftsverarbeitung von WM_OCEVENT benutzt werden.

```
//
// OC_VIEWxxxx-Botschaften verteilen
//
long
ViewWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    // insert an event cracker for each OC_VIEWxxxx message you
    // want to handle
    HANDLE_OCF(hwnd, OC_VIEWPARTINVALID, ViewWnd_OnOcViewPartInvalid);
  }
  return true;
}
```

Die WM_OCEVENT-Botschaft enthält eine Ereignis-ID in ihrem *wParam*, genau wie WM_COMMAND-Botschaften Kommando-IDs enthalten. OC_VIEWCLOSE ist eines der möglichen Ereignisse, das anzeigt, daß es an der Zeit ist, diese Ansicht des Verbunddokuments zu schließen. In Anwendungen, die nur eine Ansicht pro Dokument zeigen, signalisiert OC_VIEWCLOSE auch das Schließen des Dokuments. Das Makro HANDLE_OCF ruft die Behandlungsroutine auf, die sie für jedes Object-

Components-Ereignis vorgesehen haben, genauso wie HANDLE_MSG die Behandlungsroutine für eine Fensterbotschaft aufruft.

CPPOCF1 bearbeitet nur die OC_VIEWCLOSE-Botschaft. Um andere Botschaften zu bearbeiten, fügen Sie ein HANDLE_OCF-Makro für jede Ereignis-ID hinzu.

Eine Liste aller ObjectComponents-Botschaften findet sich in den Tabellen 20.5 und 20.6.

Bearbeiten ausgewählter View-Ereignisse

Jedes HANDLE_OCF-Makro ruft eine andere Behandlungsfunktion auf. Im folgenden Beispiel heißt die Behandlungsfunktion *ViewWnd_OnOcViewPartInvalid*. ObjectComponents sendet diese Botschaft zu einem Container, wenn eines der OLE-Datenobjekte neu aufgebaut werden muß.

```
bool
ViewWnd_OnOcViewPartInvalid(HWND hwnd, TOcPart& part)
{
  HDC dc = GetDC(hwnd);
  SetMapMode(dc, MM_ANISOTROPIC);
  SetWindowOrg(dc, 0, 0);
  SetViewportOrg(dc, 0, 0);
  RECT rect = part.GetRect();
  LPtoDP(dc, (POINT*)&rect, 2);
  InvalidateRect(hwnd, &rect, true);
  ReleaseDC(hwnd, dc);
  return true;
}
```

Der Parameter *TOcPart* steht für das Objekt, das neu aufgebaut werden muß. ObjectComponents erzeugt ein *TOcPart*-Objekt für jedes gelinkte oder eingebettete Objekt in einem Container-Dokument. CPPOCF1 bearbeitet diese Botschaft, indem es das Objekt nach seinen Koordinaten fragt und den entsprechenden Bereich seines Client-Bereichs ungültig macht. Das Kommando *InvalidateRect* resulitert in einer WM_PAINT-Botschaft, und die *ViewWnd_OnPaint*-Prozedur antwortet dadurch, daß sie das Dokument zeichnet.

Zeichnen des Dokuments

Das Zeichnen eines Dokuments erfordert zwei Schritte: Das Zeichnen der eigenen Daten des Containers und das Zeichnen aller gelinkten oder eingebetteten Objekte. Im folgenden finden Sie das Gerüst für eine Zeichenprozedur:

```
void
ViewWnd_OnPaint(HWND hwnd)
{
  PAINTSTRUCT ps;
  HDC dc = BeginPaint(hwnd, &ps);
  //
```

```
  // Zeichnen hier ausführen
  //

  // Eingebettete Objekte hier zeichnen
  ViewWnd_PaintParts(hwnd, dc, false);
  EndPaint(hwnd, &ps);
}
```

Der Code für *ViewWnd_PaintParts* ist der gleiche in den meisten Anwendungen.

```
bool
ViewWnd_PaintParts(HWND hwnd, HDC dc, bool metafile)
{
  // logische Koordinaten des Zeichnenbereichs anfordern
  TRect clientRect;
  GetClientRect(hwnd, &clientRect);
  TRect logicalRect = clientRect;
  DPtoLP(dc, (POINT*)&logicalRect, 2);

  // durch alle Teile laufen und jedes Objekt zeichnen
  ViewData& viewData = GetViewData(hwnd);
  for (TOcPartCollectionIter i(viewData.OcDoc->GetParts()); i; i++) {
    TOcPart& part = *i.Current();
    if (part.IsVisible(logicalRect)) {
      TRect rect = part.GetRect();
      part.Draw(dc, rect, clientRect, asDefault);
      if (metafile)
        continue;

      // Zeichnen eines Objekts auf verschiedene Weise,
      // je nachdem, ob es angewählt ist oder nicht
      if (part.IsSelected()) {
        // Zeichne ein XOR Rechteck um 'rect'
      }
    }
  }
  return true;
}
```

CPPOCF1 ist ein sehr einfacher Container. Da er nur jeweils ein eingebettetes Objekt enthält, muß er keine Schleife erzeugen, um mehrere Teile bearbeiten zu können. Falls er dahingehend erweitert würde, daß er mehrere Objekte zuläßt, müßte er seine Zeichnenprozedur nicht ändern.

Die Klasse *TOcPart* verwaltet gelinkte oder eingebettete Objekte in einem Container-Dokument. Die *TOcPart::Draw*-Methode fordert den Server auf, sein Objekt zu zeichnen. Der *Draw*--Methode muß die Position des Objekts nicht angegeben werden. *TOcPart* erhält diese Information, wenn

sie konstruiert wird, wie Sie im Abschnitt »Bearbeiten ausgewählter Anwendungsereignisse« sehen können.

Die for-Schleife erzeugt ein Iteratorobjekt, um alle Teile im Dokument aufzuzählen. Der Operator ++ bewegt den Iterator, um auf aufeinanderfolgende Teile zu zeigen. Der Ausdruck **i.Current()* wird jedes Mal, wenn er die Schleife wiederholt, auf einen anderen Teil gesetzt.

Aktivieren und Deaktivieren von Objekten

Nachdem ein Objekt in ein Verbunddokument eingebettet wurde, kann der Benutzer sich entschließen, das Objekt zu bearbeiten. In den meisten Containern aktiviert der Benutzer ein Objekt durch Doppelklick darauf. CPPOCF1 unterstützt keine Aktivierung von Objekten, aber der Code, dies zu erreichen, ist einfach. Sie fangen WM_LBUTTONDBLCLK-Botschaften ab, prüfen die Mauskoordinaten, und wenn diese auf ein Objekt fallen, aktivieren Sie dieses.

Verwenden Sie eine for-Schleife mit einem *TOcPartCollectionIter*-Objekt, um die eingebetteten und gelinkten Objekte eines Dokuments aufzuzählen, wie es auch die Zeichnenprozedur macht, um alle Teile zu zeichnen. Rufen Sie *TOcPart::GetRect* auf, um die Koordinaten eines Objekts zu finden. Rufen Sie *TOcPart::Activate* auf, um einen Teil zu aktivieren.

Bevor ein Container ein Verbunddokument schließt, sollte er es stets daraufhin prüfen, daß keine Objekte aktiviert sind. CPPOCF1 bindet diese Schleife in der Behandlungsroutine WM_DESTROY für sein Sichtfenster ein:

```
for (TOcPartCollectionIter i(OcDoc->GetParts()); i; i++) {
  TOcPart& p = *i.Current();
  p.Activate(false);
}
```

Die Übergabe von False an *Activate* beendet jede Bearbeitungs-Session.

3. Programmieren des Hauptfensters

Das Hauptfenster verwaltet die Aufgaben, die sich auf ein einzelnes Dokument beziehen. Es öffnet und schließt das Dokument und zeichnet es auf dem Bildschirm. Das Rahmenfenster verwaltet Aufgaben der gesamten Anwendung. Es antwortet auf Menübefehle und erzeugt und löscht das Sichtfenster.

Erzeugen des Hauptfensters

Wenn die Anwendung ihr Hauptfenster erzeugt, muß sie das Fenster an ihr *TOcApp*-Objekt binden. (Das *TOcApp*-Objekt wurde in der Factory Callback-Funktion erzeugt.)

```
bool
MainWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  HwndMain = hwnd;
```

```
  if (OcApp)
    OcApp->SetupWindow(hwnd);
  return true;
}
```

Das *TOcApp*-Objekt sendet Botschaften über OLE-Ereignisse an das Hauptfenster der Anwendung. *SetupWindow* informiert die *TOcApp* darüber, wohin sie ihre Ereignisbotschaften senden soll.

Behandlung von WM_OCEVENT

TOcApp schickt Ereignisbenachrichtigungen in der Botschaft WM_OCEVENT. Wie das Sichtfenster muß das Rahmenfenster auch WM_OCEVENT behandeln. Das Rahmenfenster erhält eine Benachrichtigung über Botschaften, die die Anwendung als Ganzes betreffen also nicht nur ein bestimmtes Dokument. Die Rahmenfensterprozedur sendet WM_OCEVENT-Botschaften an einen Behandlungsroutine, der das Ereignis identifiziert und die entsprechende Behandlungsroutine aufruft. Beide Routinen sind dem entsprechenden Code für das Sichtfenster sehr ähnlich.

```
// Standard Botschaftsbehandlungsroutine für das Hauptfenster
long CALLBACK _export
MainWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
      // andere Botschafts-Cracker gehören hierher
    HANDLE_MSG(hwnd, WM_OCEVENT,  MainWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}

// Subdispatch OC_...-Botschaften
long
MainWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    HANDLE_OCF(hwnd, OC_VIEWTITLE, MainWnd_OnOcViewTitle);
  }
  return true;
}
```

Behandlung von ausgesuchten Anwendungsereignissen

Das einzige Ereignis von ObjectComponents, das CPPOCF1 in seinem Hauptfenster behandelt, ist OC_VIEWTITLE. Diese Botschaft wird geschickt, wenn ein Server, der momentan ein Objekt offen bearbeitet, den Titel des Containers in seinem eigenen Fenstertitel verwenden möchte. Die Richtlinien für die OLE Anwenderschnittstelle verlangen, daß der Server die Quelle des bearbeiteten Objekts anzeigt.

```
const char*
MainWnd_OnOcViewTitle(HWND /*hwnd*/)
{
```

```
  return APPSTRING;
}
```

Da CPPOCF1 immer nur ein einziges Dokument besitzt, gibt es den Namen der Anwendung als den Titel seiner Ansicht zurück. Da es immer nur eine einzige Ansicht besitzt, kann es das Ereignis OC_VIEWTITLE in der Hauptfensterprozedur behandeln. Die meisten Container behandeln diese Botschaft im Sichtfenster und geben den Namen der Anwendung und den Namen des Dokuments in einer einzigen Zeichenkette zurück.

Behandlung von Standard-OLE-Menübefehlen

Ein OLE-Container besitzt in seinem Menü *Editieren* OLE-Befehle. Tabelle 20.2 beschreibt die neuen Befehle. Es ist nicht notwendig, alle Befehle zu verwenden. CPPOCF1 unterstützt den Befehl Objekt Einfügen. Dieser Befehl erlaubt dem Anwender, neue Objekte dem aktuellen Dokument hinzuzufügen.

```
void
MainWnd_OnCommand(HWND hwnd, int id, HWND /*hwndCtl*/, uint
/*codeNotify*/)
{
  switch (id) {
    case CM_INSERTOBJECT: {
      try {
        TOcInitInfo initInfo(OcView);           // Mit der
                                                // Initialisierung der
                                                //  Info-Struktur beginnen
        if (OcApp->Browse(initInfo)) {          // Dialogfeld Objekt
                                                // einfügen anzeigen
          TRect rect(30, 30, 100, 100);         // nur oben und links
                                                // erden verwendet
          new TOcPart(*OcDoc, initInfo, rect);  // neues Objekt zum
                                                // Dokument hinzufügen
        }
      }
      catch (TXBase& xbase) {
        MessageBox(GetFocus(), xbase.why().c_str(), "Exception caught",
MB_OK);
      }
      break;
    }
    case CM_EXIT: {
      PostMessage(hwnd, WM_CLOSE, 0, 0);
      break;
    }
  }
}
```

Der Code zum Einfügen oder Ablegen eines Objekts in ein Dokument beginnt immer mit einer Struktur *TOcInitInfo.* enthält Information, die das

zu erzeugende Objekt beschreibt: welcher Container es empfangen wird, ob es gelinkt oder eingebettet wird, ob es schon vorhanden ist oder neu erzeugt wird, ob es sich dort befindet, wo die Daten sind und in welchem Format es vorliegt.

Der Konstruktor für *TOcInitInfo* erhält einen Zeiger auf die Ansicht, in der Sie das neue Objekt darstellen möchten. Der nächste Befehl, *OcApp->Browse* ruft das Standard-Dialogfeld *Objekt einfügen* auf, das dem Anwender eine Auswahl aller Objekte anbietet, die ein beliebiger Server, der im System registriert ist, erzeugen kann. Wenn der Anwender eins auswählt, setzt der Befehl *Browse* mehr Information in *initInfo*.

Der letzte Schritt, um ein neues OLE-Objekt einzufügen, besteht darin, einen *TOcPart*-Verbinder zu erzeugen. *TOcPart* implementiert alle OLE-Dienste, die von einem gelinkten oder eingebetteten Objekt verlangt werden. Es verbindet sich mit OLE, holt die Informationen für ein neues Objekt, setzt sich auf die Liste von Teilen in *OcDoc* und stellt sich auf dem Bildschirm an der durch *TRect* bestimmtenPosition dar.

Beispiele, die den Einsatz anderer OLE-Menüeditierbefehle zeigen finden Sie im Quelltext für die Ereignisbehandlungsroutine in OWL/OLEWINDO.CPP.

Erzeugen des Programms

Um den Server zu erzeugen, müssen Sie die richtigen Header-Dateien einsetzen, ein unterstütztes Speichermodell verwenden und die richtigen Bibliotheken linken.

Einbinden der ObjectComponents Header-Dateien

Die folgende Liste zeigt die ObjectComponents Header-Dateien für einen Container, der nicht ObjectWindows verwendet.

```
#include <ocf/ocapp.h>       // TOcRegistrar, TOcModule, TOcApp
(application connector)
#include <ocf/ocdoc.h>       // TOcDocument (Verbunddokument-Manager)
#include <ocf/ocview.h>      // TOcView (Dokumentansichtverbinder)
#include <ocf/ocpart.h>      // TOcPart (Verbinder für
                                gelinkte/eingebettete Objecte)
#include <ocf/ocfevx.h>      // WM_OCEVENT Botschafts-Cracker
```

Compilieren und linken

ObjectComponents-Anwendungen, die nicht ObjectWindows verwenden, können entweder das Speichermodell medium oder large verwenden. Linken Sie diese mit den OLE- und ObjectWindows-Bibliotheken.

Um CPPOCF0, CPPOCF1, und CPPOCF2 zu erzeugen, gehen Sie in das Verzeichnis des Programms und geben Sie folgendes in der Kommandozeile ein:

```
make MODEL=l
```

Dieser Befehl erzeugt alle drei Programme mit dem Speichermodell large.

Die Make-Datei, die dieses Beispiel erzeugt, bezieht sich auf die Datei OCFMAKE.GEN.

K A P I T E L 21

Erzeugen eines OLE-Servers

Ein OLE-Server ist eine Anwendung, die Datenobjekte für andere Programme erzeugt und verwaltet. Dieses Kapitel erläutert, wie man vorhandene Programme in OLE-Server mit Verknüpfungs- und Einbettungsfähigkeiten umwandelt. Dabei werden die für die drei folgenden Programmtypen erforderlichen Schritte beschrieben:

- Eine ObjectWindows-Anwendung, die das Dok/View-Modell benutzt.
- Eine ObjectWindows-Anwendung, die das Dok/View-Modell nicht benutzt.
- Eine C++ Anwendung, die ObjectWindows nicht benutzt.

Außerdem behandelt dieses Kapitel folgende verwandte Themen:

- Registriertabellen
- DLL-Server

Die Gerüstklassen von ObjektComponents unterstützen sowohl Server als auch Container, neue ObjektWindows-Klassen machen die Nutzung dieser Unterstützung einfach. Der am einfachsten umzuwandelnde Programmtyp ist derjenige, der ObjectWindows und das Dok/View-Modell verwendet, aber ObjektComponents vereinfacht sogar die Aufgabe des Schreibens einer Server-Anwendung ohne das ObjectWindows-Gerüst.

Umwandeln einer Dok/View-Anwendung in einen OLE-Server

Zum Umwandeln einer Dok/View-Anwendung in einen OLE-Server sind nur wenige Änderungen erforderlich; viele davon gleichen denen, die zum Erzeugen eines Containers erforderlich sind. Falls Sie Ihre Anwendung bereits gemäß den Schritten in Kapitel 20 modifiziert haben, ist ein großer Teil Ihrer Arbeit zur Erzeugung eines Servers bereits erledigt.

Die folgende Liste zeigt die Änderungen in Kurzform. In den darauffolgenden Abschnitten wird jeder Schritt im Detail beschrieben.

1. Verbinden Sie Ihre Anwendungs-, Fenster-, Dokument- und View-Objekte mit dem OLE-System.
   - Erzeugen Sie ein Objekt der Klasse *TAppDictionary*.
   - Leiten Sie Ihre Anwendungsklasse von *TOcModule* ab, ebenso wie *TApplication*.
   - Leiten Sie die Rahmenfenster-, Dokument und View-Klassen von den neuen OLE-fähigen Klassen ab.
2. Registrieren Sie die Anwendung.
   - Erzeugen Sie Registriertabellen, die die Anwendung und die Typen der von ihr erzeugten Dokumente beschreiben.
   - Erzeugen Sie ein Objekt *Registrar*, und rufen Sie dessen Methode *Run* auf.
3. Stellen Sie die Koordination mit dem Container her, um Ihre Objekte grafisch darstellen, laden und speichern zu können.
   - Melden Sie an das OLE-System, wenn sich das Aussehen eines Objekts ändert. Dazu ist nur ein einfacher Funktionsaufruf nötig.
   - Fügen Sie Befehle zum Lesen und Schreiben von eingebetteten Objekten in Verbunddokumenten hinzu.
4. Erzeugen Sie die Anwendung.
   - Setzen Sie am Anfang Ihres Quelltexts Include-Anweisungen für die neuen Include-Dateien des ObjektWindows-OLE-Systems.
   - Compilieren Sie im Speichermodell Large. Linken Sie dann mit den Bibliotheken des OLE-Systems und von ObjektComponents.

Das ist schon alles, was Sie tun müssen. Nach Durchführung dieser Schritte unterstützt Ihr OLE-Server alle folgenden Funktionen:

- Quelle für Verknüpfungen
- Quelle für Einbettungen
- Quelle für Ziehen-und-Ablegen
- Bearbeitung in der Zielanwendung
- Registrierung
- Speicherung von Verbunddokumenten

Um das voreingestellte Verhalten ändern zu können, das ObjektComponents für die gängigen OLE-Optionen vorsieht, können Sie zusätzlich die standardmäßigen Behandlungsroutinen für die von ObjektComponents gesendeten Botschaften außer Kraft setzen. Für eine Liste der Ereignisbotschaften siehe die Tabellen 19.5 und 19.6.

1. Verbinden von Objekten mit dem OLE-System

Ihre Anwendungs-, Fenster-, Dokument- und View-Objekte müssen die neuen OLE-fähigen Klassen benutzen. Der Konstruktor für das Anwendungsobjekt erwartet die Übergabe eines Anwendungsverzeichnisobjekts, erzeugen Sie dieses deshalb zuerst.

Erzeugen eines Anwendungsverzeichnisses

Ein Anwendungsverzeichnis enthält laufend aktualisierte Informationen für den gerade aktiven Prozeß. Es ist besonders für DLLs nützlich. Wenn mehrere Prozesse eine DLL gemeinsam benutzen, so muß diese mehrere Kopien der globalen, statischen und dynamischen Variablen verwalten, die den aktuellen Zustand der DLL im jeweiligen Prozeß repräsentieren. Beispiel: Die DLL-Version von ObjectWindows verwaltet ein Verzeichnis, mit dem der DLL ermöglicht wird, dasjenige Objekt der Klasse *TApplication* abzurufen, das zu dem gerade aktiven Client-Prozeß gehört. Wenn Sie einen ausführbaren Server in einen DLL-Server umwandeln, so muß letzterer ein Verzeichnis der Objekte der Klasse *TApplication* verwalten, von denen jedes einen seiner Container Clients repräsentiert.

Das Makro DEFINE_APP_DICTIONARY stellt einen einfachen und vereinheitlichten Weg zur Erzeugung des Verzeichnisobjekts für jede Art von Anwendung bereit, ganz gleich, ob es ein Container, ein Server, eine DLL oder ein EXE-Programm ist. Fügen Sie dazu die folgende Anweisung bei Ihren Deklarationen der anderen statischen Variablen ein:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

Das Makro erzeugt einfach für jede zu der statischen Version der DLL-Bibliothek hinzugelinkten Anwendung eine Referenz auf das Anwendungsverzeichnis in ObjectWindows. Für DLL-Server, die die DLL-Version ObjectWindows benutzen, erzeugt es statt dessen eine Instanz der Klasse *TAppDictionary*.

Hinweis

Geben Sie Ihrem Verzeichnisobjekt den Namen *AppDictionary*, um die Vorteile von Hersteller-Templates wie etwa *TOleDocViewFactory* (wie später erklärt unter »Erzeugen eines Registrar-Objekts«) zu nutzen.

Ableiten des Anwendungsobjekts von TOcModule

Das Anwendungsobjekt eines ObjektComponents-Programms muß ebenso wie *TApplication* von *TOcModule* abgeleitet werden. *TOcModule* koordiniert einige grundlegende Verwaltungsaufgaben im Zusammenhang mit Registrierung und Speicherverwaltung. Außerdem verbindet es Ihre Anwendung mit dem OLE-System. Genauer gesagt verwaltet *TOcModule* das Verbinderobjekt, das COM-Schnittstellen für eine Anwendung implementiert.

Angenommen, die Deklaration Ihres Anwendungsobjekts sieht so aus:

```
class TMyApp : public TApplication {
  public:
```

```
    TMyApp() : TApplication(){};
  ⋮
};
```

Ändern Sie sie dann in folgende Form:

```
class TMyApp : public TApplication, public TOcModule{
   public:
      TMyApp(): TApplication(::AppReg["description"], ::Module,
                             &::AppDictionary){};
  ⋮
   };
```

Dem Konstruktor für die geänderte Klasse *TMyApp* werden drei Parameter übergeben:

- Ein String mit dem Namen der Anwendung

AppReg ist die Registriertabelle der Anwendung, siehe dazu unten unter »2. Registrieren eines OLE-Servers«. Der Ausdruck *::AppReg["description"]* extrahiert einen String, der zur Beschreibung der Anwendung registriert wurde.

- Ein Zeiger auf das Anwendungsmodul.

Module ist eine globale Variable des Typs *TModule**, definiert durch ObjectWindows.

- Die Adresse des Anwendungsverzeichnisses.

AppDictionary ist das Anwendungsverzeichnisobjekt, das im vorigen Abschnitt beschrieben wurde.

Erben von OLE-Klassen

Ein Server nimmt an seinen OLE-Klassen dieselben Veränderungen vor, wie es ein Container tut. ObjectWindows enthält in seinen neuen Fenster-, Dokument- und View-Klassen sehr leistungsfähige Funktionen. Wenn Sie einem ObjectWindows-Programm OLE-Fähigkeiten geben wollen, so ändern sie seine abgeleiteten Klassen so ab, daß sie von OLE-Klassen erben.

Hier einige Beispiele:

```
// alte Deklarationen (ohne OLE)
class TMyDocument: public TDocument    { /* Deklarationen */ };
class TMyView: public TView                    { /* Deklarationen */ };
class TMyFrame: public TFrameWindow   { /* Deklarationen */ );

// neue Deklarationen (mit OLE)
class TMyDocument: public TOleDocument { /* Deklarationen */ };
class TMyView: public TOleView          { /* Deklarationen */ };
class TMyFrame: public TOleFrame       { /* Deklarationen */ );
```

Für eine vollständige Liste der neuen OLE-Klassen und ihrer Nicht-OLE-Gegenstücke siehe Tabelle 20.1.

Wenn Sie auf OLE-Klassen umsteigen, so vergewissern Sie sich, daß die Methoden in Ihren Klassen, die Bezüge auf ihre direkten Basisklassen enthalten, nunmehr die Namen der OLE-Klassen benutzen.

```
void TMyView::Paint(TDC& dc, BOOL erase, TRect& rect)
{
   TOleView::Paint(dc, erase, rect);
   // Erzeugen Sie hier die grafische Darstellung
}
```

Es ist grundsätzlich sicherer, es den OLE-Klassen zu erlauben, die Windows Ereignisse und Dok/View-Botschaften zu behandeln. Dies gilt besonders für die Methode *Paint* und für Behandlungsroutinen für Mausbotschaften in einer von *TOleView* abgeleiteten Klasse. *TOleView::Paint* weiß, wie die in Ihrem Dokument eingebetteten Objekte grafisch dargestellt werden müssen. (Server sind oft zugleich Container, und in das Objekt eines Servers sind möglicherweise andere Objekte eingebettet.) In ähnlicher Weise ermöglichen die Mausroutinen von *TOleView* es dem Benutzer, ein OLE-Objekt, das mit Ihrem Dokument verknüpft oder darin eingebettet ist, zu markieren, zu verschieben, in der Größe zu verändern und zu aktivieren.

2. Registrieren eines OLE-Servers

Um Ihre OLE-fähige Anwendung zu registrieren, müssen Sie zuerst Registriertabellen erzeugen, die Ihre Anwendung und die Typen der von ihr erzeugten Dokumente beschreiben. Die Tabellen erzeugen Strukturen, die dem Registrar-Objekt übergeben werden, wenn Sie dieses erzeugen. Um die Anwendung zu starten, rufen Sie die Methode *Run* des Registrar-Objekts auf.

Erzeugen von Registriertabellen

Server implementieren OLE-Objekte, die jeder Container benutzen kann. Unterschiedliche Server implementieren unterschiedliche Typen von Objekten. Jeder Typ von Objekt, den ein Server erzeugt, muß einen global eindeutigen Namen (GUID) und einen eindeutigen String-Bezeichner haben. Jeder Server muß diese Information zusammen mit anderen beschreibenden Informationen in die Registrierungsdatenbank des Systems, unter dem er läuft, schreiben. Das OLE-System liest die Registrierungen, um zu ermitteln, welche Objekte verfügbar sind, welche Fähigkeiten sie haben und wie die Anwendung aufgerufen werden muß, die Objekte eines bestimmten Typs erzeugt.

ObjektComponents vereinfacht die Aufgabe der Registrierung. Sie rufen einfach Makros auf, um eine Tabelle von Schlüsseln mit zugehörigen Werten zu erzeugen. ObjektComponents bekommt die Tabelle übergeben und erledigt dann automatisch alle Registrierungsaufgaben.

```
REGISTRATION_FORMAT_BUFFER(100)         // Platz zum Erweitern der Makros
reservieren
```

```
BEGIN_REGISTRATION(AppReg)
   REGDATA(clsid,       "{5E4BD320-8ABC-101B-A23B-CE4E85D07ED2}")

   REGDATA(description,"Server für Drawing Pad")
END_REGISTRATION

BEGIN_REGISTRATION(DocReg)
   REGDATA(progid,      "DrawPad.Document.15")
   REGDATA(description,"Drawing Pad (Step15--Server)")
   REGDATA(menuname,    "Drawing Pad 15")
   REGDATA(extension,   "p15")
   REGDATA(docfilter,   "*.p15")
   REGDOCFLAGS(dtAutoOpen | dtAutoDelete | dtUpdateDir | dtCreatePrompt |
dtRegisterExt)
   REGDATA(insertable, "")
   REGDATA(verb0,       "&Edit")
   REGDATA(verb1,       "&Open")
   REGFORMAT(0, ocrEmbedSource,  ocrContent,  ocrIStorage,  ocrGet)
   REGFORMAT(1, ocrMetafilePict, ocrContent,  ocrMfPict|ocrStaticMed,
ocrGet)
   REGFORMAT(2, ocrBitmap,       ocrContent,  ocrGDI|ocrStaticMed, ocrGet)
   REGFORMAT(3, ocrDib,          ocrContent,  ocrHGlobal|ocrStaticMed,
ocrGet)
   REGFORMAT(4, ocrLinkSource,   ocrContent,  ocrIStream, ocrGet)
END_REGISTRATION
```

Die Makros des Beispiels erzeugen zwei Strukturen. Die erste Struktur heißt *AppReg*, die zweite *DocReg*. ObjektComponents benutzt Strings mit Kleinbuchstaben wie *progid* und *clsid* als Namen für die Standardschlüssel, denen Sie Werte zuordnen müssen. Die von Ihnen zugeordneten Werte sind Strings, bei denen die Groß/Kleinschreibung signifikant ist. Die Reihenfolge der Schlüssel in der Registriertabelle ist ohne Bedeutung. Für weitere Informationen über die Makros REGDATA, REGFORMAT und REGDOCFLAGS siehe unter »Registriermakros«. Die gesamte Gruppe der Registriermakros wird im *ObjectWindows Referenzhandbuch* beschrieben.

Die Gruppe von Schlüsseln, die Sie in eine Struktur einsetzen, hängt davon ab, welche OLE-Fähigkeiten Sie unterstützen wollen, und davon, ob die Struktur Anwendungs- oder Dokumentinformationen enthält. Die Makroaufrufe des Beispiels zeigen die Mindestmenge an Information, die ein Server mit einem einzigen Dokumenttyp bereitstellen sollte.

Ein Server registriert Programm- und Klassen-ID-Strings (*progid* und *clsid*) für sich selbst und für jeden Typ von Dokumenten, die er erzeugt. Die IDs müssen unter allen Umständen eindeutig sein, damit das OLE-System die einzelnen Anwendungen voneinander unterscheiden kann. Die Strings *description* erscheinen im Dialogfenster *Einfügen/Objekt*, das dem Benutzer eine Liste der im System verfügbaren Objekte zeigt.

Tabelle 21.1 beschreibt in Kurzform alle Registrierschlüssel, die ein OLE-Server benutzen kann. Dabei wird bei jedem Schlüssel angegeben, ob er optional oder zwingend erforderlich ist, und ob er in die Anwendungs- oder in die Dokumentregistriertabelle gehört.

Tabelle 21.1: Schlüssel, die ein OLE-Server registriert

Schlüssel	In *AppReg*?	In *DocReg*?	Beschreibung
clsid	Ja	Optional	Global eindeutiger Bezeichner (GUID); automatisch für die Struktur DocReg erzeugt
description	Ja	Ja	Beschreibungs-String (bis zu 40 Zeichen)
progid	Nein	Ja, bei verknüpfter oder eingebetteter Quelle	Bezeichner für den Typ des Programms oder Objekts (eindeutiger String)
menuname	Nein	Ja	Name des Serverobjekts für das Containermenü
extension	Nein	Optional	Namenserweiterung des Dokuments, das dem Server zugeordnet ist
docfilter	Nein	Ja, falls nicht dtHidden	Dateinamen mit Jokerzeichen für das Dialogfenster Datei/Öffnen
docflags	Nein	Ja	Optionen für das Starten des Dialogfensters Datei/Öffnen
debugger	Nein	Optional	Kommandozeile zum Starten des Debuggers
debugprogid	Nein	Optional	Name der Debuggerversion (eindeutiger String)
debugdesc	Nein	Ja, falls debugprogid benutzt wird	Beschreibung der Debuggerversion
insertable	Nein	Ja, für Einbettungsfunktion	Gibt an, daß das Objekt eingebettet werden kann. Fehlt der Schlüssel, so ist das Dokument nur eine Verknüpfungsquelle
verb*n*	Nein	Ja	Eine Aktion, die der Server mit dem Objekt durchführen kann
format*n*	Nein	Ja	Ein Datenformat für die Zwischenablage, das der Server liefern kann.
aspectall	Nein	Optional	Optionen zur Anzeige des Objekts in jedem Aspekt
aspectcontent	Nein	Optional	Optionen zur Anzeige des Inhaltsaspekts des Objekts
aspectdocprint	Nein	Optional	Optionen zur Anzeige des Druckaspekts des Objekts
aspecticon	Nein	Optional	Optionen zur Anzeige des Icon-Aspekts des Objekts
aspectthumbnail	Nein	Optional	Optionen zur Anzeige des Miniaturbildaspekts des Objekts
cmdline	Nein	Optional	Argumente, die in die Kommandozeile des Servers gesetzt werden
path	Nein	Optional	Pfad zur Server-Datei (Voreinstellung: Pfad des aktuellen Moduls)
permid	Nein	Optional	Namens-String ohne Versionsinformation
permname	Nein	Optional	Beschreibungs-String ohne Versionsinformation
usage	Optional	Optional	Unterstützung für mehrere Clients gleichzeitig
language	Optional	Nein	Sprache für registrierte Strings (Voreinstellung: vom Benutzer gewählte Systemeinstellung)
version	Optional	Nein	Haupt- und Unterversionsnummern (Voreinstellung: "1.0")

Die Tabelle geht davon aus, daß die Dokumente des Servers Verknüpfung oder Einbettung unterstützen. Bei Dokumenten, für die beides nicht der Fall ist, braucht der Server nur *docflags* und *docfilter* zu registrieren.

Falls der Server zugleich ein Container oder ein Automatisierungs-Server ist, sollten Sie sich auch die Container-Tabelle (Tabelle 20.3) oder die Automatisierungstabelle (Tabelle 22.1) ansehen. Registrieren Sie alle Schlüssel, die in einer der Tabellen erforderlich sind, die zu Ihrer Anwendung gehört.

Mehr Informationen über einzelne Registrierungsschlüssel, die Werte, die sie annehmen können und die Makros, mit deren Hilfe man sie registriert, finden Sie im *ObjectWindows Referenzhandbuch* unter dem Namen des jeweiligen Schlüssels.

Die den Schlüsseln zugewiesenen Werte können übersetzt werden, um sie an die in der Systemsteuerung eingestellte Sprache anzupassen. Mehr über Lokalisierung finden Sie im Abschnitt »Registrieren lokalisierter Einträge« und im Abschnitt »Lokalisieren von Symbolnamen«.

Schreiben Sie Ihre Registrierstrukturen dort in die Quelltextdateien, wo sie Dokument-Templates aufbauen und ihre von *TApplication* abgeleitete Klasse implementieren. Ein Server erzeugt immer nur eine Anwendungsregistriertabelle (im Beispiel *AppReg* genannt). Ein Server kann jedoch mehrere Dokumentregistriertabellen erzeugen, falls er mehrere unterschiedliche Dokumenttypen erzeugen kann (z.B. Textobjekte und Tabellenkalkulationsobjekte). Jede Registriertabelle muß einen eindeutigen Wert für *progid* haben.

Nachdem Sie Ihre Registriertabellen erzeugt haben, müssen Sie sie den richtigen Objektkonstruktoren übergeben. Die Struktur *AppReg* wird dem Konstruktor *TOcApp* übergeben, wie in Abschnitt »Erzeugen eines Registrar-Objekts« beschrieben. Die Dokumentregistriertabellen werden dem Konstruktor des Dokument-Templates übergeben.

```
DEFINE_DOC_TEMPLATE_CLASS(TMyOleDocument, TMyOleView, MyTemplate);
MyTemplate myTpl(::DocReg);
```

Ein Teil der Informationen in der Dokumentregistriertabelle wird nur vom Dokument-Template benutzt. Die Dokumentfilter und Dokument-Flags haben nur etwas mit dem Dokument zu tun, nicht mit den OLE-Funktionen. Frühere Versionen von OWL haben die gleichen Informationen an das Dokument übergeben, und zwar in Form einer Reihe einzelner Parameter. Diese alte Methode wird zur Rückwärtskompatibilität noch unterstützt, aber neue Programme sollten, ganz gleich ob sie OLE-Funktionen benutzen oder nicht, die Registriermakros zur Übergabe der Parameter für das Dokument-Template benutzen.

Einige der Registriermakros erweitern die ihnen übergebenen Werte. Das Makro REGISTRATION_FORMAT_BUFFER reserviert für die Erweiterung temporären Speicher. Um zu ermitteln, wieviel Pufferspeicher Sie benötigen, setzen Sie 10 Bytes für jeden Eintrag des Typs REGFORMAT plus der Länge aller String-Parameter, die sie an die Makros REGSTATUS, REGVERBOPT, REGICON oder REGFORMAT übergeben. (Die Registriermakros werden im *ObjectWindows Referenzhandbuch* beschrieben.)

Erzeugen eines Registrar-Objekts

Jede ObjektComponents-Anwendung benötigt ein Registrar-Objekt zur Verwaltung ihrer Registrierungsaufgaben. In einer Anwendung, die OLE-Möglichkeiten bietet, hat das Registrar-Objekt den Typ *TOcRegistrar*. Deklarieren Sie am Anfang Ihrer Quelltextdatei eine globale Variable, die einen Zeiger auf das Registrar-Objekt aufnimmt:

```
static TPointer<TOcRegistrar> Registrar;
```

Das Template *TPointer* stellt sicher, daß die Instanz der Klasse *TOcRegistrar* gelöscht wird, wenn das Programm endet.

Hinweis

Geben Sie dieser Variablen den Namen *Registrar*, um die Vorteile zu nutzen, die das Callback Template des Herstellers im Konstruktor des Registrars bietet.

Der nächste Schritt besteht darin, Ihre Funktion *OwlMain* so zu modifizieren, daß sie ein neues Objekt der Klasse *TOcRegistrar* erzeugt und die globale Zeigervariable *Registrar* initialisiert. Der Konstruktor von *TOcRegistrar* erwartet drei Parameter: die Anwendungsregistrierstruktur, die Callback-Funktion des Herstellers der Komponente und den Kommandozeilen-String, mit dem die Anwendung aufgerufen wurde. Die Registrierstruktur wird mit Hilfe der Registriermakros erzeugt.

Die Callback-Funktion des Herstellers wird mit einer Template-Klasse erzeugt. Bei einer ObjektWindows-Anwendung mit OLE-Funktionen, die Dok/View benutzt, heißt die Template-Klasse *TOleDocViewFactory*. Der dritte Parameter, der Kommandozeilen-String, kann von der Methode *GetCmdLine* der Klasse *TApplication* geliefert werden. Der Code im Hersteller-Template geht davon aus, daß Sie ein Anwendungsverzeichnis mit dem Namen *AppDictionary* und einen Zeiger *TOcRegistrar** mit dem Namen *Registrar* definiert haben.

```
int OwlMain(int, char*[])
{
   // Erzeugen des Registrar-Objekts
   ::Registrar = new TOcRegistrar(::AppReg, TOleDocViewFactory<TMyApp>(),
         TApplication::GetCmdLine());
   .
   .
....
}
```

Nach dem Initialisieren des Zeigers *Registrar* muß Ihre OLE-Container-Anwendung die Methode *Run* des Registrar-Objekts an Stelle von *TApplication::Run* aufrufen. *TRegistrar::Run* ruft die Callback-Funktion des Herstellers (auf die der zweite Parameter zeigt) auf und bewirkt, daß sich die Anwendung selbst erzeugt. Die Anwendung geht dann in die Ereignisschleife, die effektiv in der Callback-Funktion des Herstellers liegt. Der folgende Code zeigt ein Beispiel für *OwlMain* vor und nach dem Hinzufügen eines Registrar-Objekts. Die Änderungen sind durch Fettdruck hervorgehoben.

Vorher:

```
// OwlMain ohne OLE-Funktionen
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
   return TMyApp().Run();
}
```

Danach:

```
// Neue Deklaration von OwlMain
int
OwlMain(int /*argc*/, char* /*argv*/[])
{
   ::Registrar = new TOcRegistrar(::AppReg,
         TOleDocViewFactory<TMyApp>(),
         TApplication::GetCmdLine());
   return ::Registrar->Run();
}
```

Der letzte Parameter des Konstruktors von *TOcRegistrar* ist der Kommandozeilen-String, mit dem die Anwendung aufgerufen wurde.

Verarbeiten der Kommandozeile

Wenn das OLE-System einen Server aufruft, setzt es das Flag *-Embedding* in die Kommandozeile, um der Anwendung den Grund ihres Aufrufs mitzuteilen. Das Vorhandensein dieses Kommandozeilenschalters zeigt an, daß der Benutzer den Server nicht direkt gestartet hat. Normalerweise reagiert der Server darauf so, daß er sein Hauptfenster unsichtbar läßt. Der Benutzer arbeitet mit dem Server über den Container. Wenn der Schalter *-Embedding* nicht vorhanden ist, so hat der Benutzer den Server als unabhängige Anwendung aufgerufen, der Server zeigt sich dann in der normalen Weise.

Wenn Sie ein Objekt *TRegistrar* per Konstruktor erzeugen, so zerlegt es die Kommandozeile für Sie und sucht nach OLE-bezogenen Schaltern. Es entfernt die Schalter, während es sie bearbeitet; wenn Sie die Kommandozeile nach dem Erzeugen des Objekts abfragen, werden Sie diese Schalter daher niemals sehen. Wenn Sie wissen wollen, welche Schalter ursprüng-

lich erkannt wurden, so rufen Sie *IsOptionSet* auf. Normalerweise *müssen* Sie auf jeden Fall *IsOptionSet* aufrufen, um festzustellen, ob der Server sichtbar gemacht werden muß.

```
if (MyApp->IsOptionSet (amEmbedding))
   nCmdShow = SW_HIDE;
```

Ihre Anwendung erbt *IsOptionSet* von *TOcModule*. Wenn das Flag *amEmbedding* gesetzt ist, so hat *TRegistrar* den Schalter *-Embedding* erkannt. In diesem Fall sollte der Server sein Element *nCmdShow* modifizieren, um das Hauptfenster unsichtbar zu halten.

Tabelle 21.2 zeigt alle OLE-bezogenen Kommandozeilenschalter.

Tabelle 21.2: Von ObjektComponents erkannte Kommandozeilenschalter

Schalter	Erwartete Reaktion des Servers	Schalter vom OLE-System generiert?
-RegServer	Registrieren aller seiner Informationen und beenden.	Nein
-UnregServer	Löschen aller seiner Einträge aus dem System und beenden.	Nein
-Automation	Sich selbst als nicht-mehrfach benutzbar (nur ein Client) registrieren. Stets zusammen mit *-Embedding*.	Ja
-Embedding	Nicht sichtbar bleiben, da er nur für ein Client und nicht als Programm läuft.	Ja
-LangID	Einstellen der Sprache für die Registrierung und den Typbibliotheken.	Nein
-TypeLib	Erzeugen und Registrieren einer Typbibliothek.	Nein
-Debug	Starten einer Debug-Sitzung.	Ja

Das OLE-System setzt einige der Schalter in die Kommandozeile des Programms ein. Jeder kann andere Flags einsetzen, um ObjektComponents zur Ausführung bestimmter Aktionen zu veranlassen. Ein Installationsprogramm könnte zum Beispiel die von ihm installierte Anwendung mit dem Schalter *-RegServer* aufrufen, so daß sich der Server selbst registriert. Ein Schalter kann entweder mit einem Bindestrich (-) oder mit einem Schrägstrich (/) beginnen.

Flags;OLE-ServerNur wenige Schalter fordern eine Aktion Ihrerseits an. Wenn ein Server oder ein Automatisierungsobjekt einen der Schalter *-Embedding* und *-Automation* erkennt, so kann er entscheiden, ob er sein Hauptfenster unsichtbar läßt. Normalerweise nimmt ObjektComponents Ihnen diese Entscheidung ab. Sie können den Schalter *-Debug* als Signal zum Aktivieren und Deaktivieren von Trace-Meldungen benutzen, aber es ist dem Server stets freigestellt, ob er auf den Schalter *-Debug* reagiert. (Das OLE-System benutzt den Schalter *-Debug* nur, wenn Sie den Schlüssel *debugprogid* registriert haben.)

ObjektComponents verwaltet alle anderen Schalter für Sie. Wenn der Benutzer ein Programm mit dem Schalter *-UnregServer* aufruft, so liest Ob-

jektComponents dessen Registriertabellen und löscht alle seine Einträge aus der Registrierungsdatenbank. Wenn ObjektComponents in der Kommandozeile mehrere Schalter findet, bearbeitet es sie alle. Das folgende Beispiel veranlaßt ObjektComponents dazu, eine Typbibliothek in der voreingestellten Sprache und eine zweite in Französisch (belgische Variante) zu erzeugen.

```
myapp -TypeLib -LangID=80C -TypeLib
```

Die als *-LangID* übergebende Zahl muß aus Sedezimalziffern bestehen. Das Win32 API definiert *80C* als Landeskennung für den belgischen Dialekt der französischen Sprache. Damit die obige Kommandozeile die beschriebene Wirkung hat, muß *myapp* natürlich Strings in belgischem Französisch in seinen XLAT-Ressourcen haben. Weitere Informationen über die Lokalisierung finden Sie unter »Lokalisieren von Symbolnamen«.

Dem Flag *-RegServer* kann optional einen Dateinamen angefügt werden:

```
myapp -RegServer = MYAPP.DAT
```

Dadurch wird ObjektComponents dazu veranlaßt, eine Registrierdatendatei MYAPP.DAT anzulegen. Diese neue Datei enthält alle Registrierinformationen der Anwendung. Wenn Sie MYAPP.DAT Ihrem Programm beilegen, so können die Benutzer sie direkt in ihre eigene Registrierungsdatenbank einfügen. Ohne Angabe eines Dateinamens bewirkt *-RegServer*, daß alle Daten direkt in die Registrierungsdatenbank des Systems geschrieben werden.

Eine Beschreibung des Schalters *-TypeLib* finden Sie unter »Erzeugen einer Typbibliothek«.

Hinweis

Nur Server, die aus einer EXE-Datei bestehen, arbeiten mit echten Kommandozeilen. An DLLs kann das OLE-System keine Kommandozeilenoptionen übergeben. ObjektComponents simuliert die Übergabe einer Kommandozeile an einen DLL-Server, so daß Sie in beiden Fällen denselben Code benutzen können. Das Registrar-Objekt setzt stets die richtigen Flags für den Ablaufmodus. Weitere Informationen über DLL-Server finden Sie unter »Erzeugen eines DLL-Servers«.

3. Darstellen, Laden und Speichern von Objekten

Ein Server muß sich mit seinem Client koordinieren, um seine Objekte dann zu bearbeiten, wenn sie dargestellt oder gespeichert werden müssen.

Mitteilung an den Client bei Änderung des Objekts

Der Server muß vorgenommene Modifikationen, die die Erscheinung eines Objekts auf dem Bildschirm verändern, dem OLE-System mitteilen. Das OLE-System speichert eine Metadateidarstellung jedes verknüpften oder eingebetteten Objekts, so daß es das Objekt für den Container selbst

dann noch darstellen kann, wenn der Server nicht aktiv ist. Wenn sich das Objekt ändert, muß das OLE-System die Metadatei aktualisieren. Der Server benachrichtigt das OLE-System von der Änderung, indem er *TOleView::InvalidatePart* aufruft. Das OLE fordert dann seinerseits den Server auf, das geänderte Objekt in eine neue Metadatei zu schreiben. ObjektComponents bearbeitet diese Anforderung durch Übergabe des Gerätekontexts der Metadatei an die Server-Funktion *Paint*. Sie müssen also keinen zusätzlichen Code schreiben, um die Metadatei zu aktualisieren.

Eine gute Stelle für den Aufruf von *InvalidatePart* ist eine Position innerhalb der Behandlungsroutinen für die Meldungen, die ObjectWindows an eine View sendet, wenn seine Daten sich ändern:

```
bool TDrawView::VnRevert(bool /*löschen*/) {
   Invalidate();              // komplette Neuanzeige erzwingen
   InvalidatePart(invView);   // Container über die Änderung informieren
   return true;
}
```

invView ist eine, von ObjectComponents definierte Aufzählungskonstante, die anzeigt, daß das View ungültig ist und erneuert werden muß.

Andere Ansichts-Benachrichtigungsbotschaften, die die Notwendigkeit einer Aktualisierung anzeigen, sind EV_VN_APPEND, EV_VN_MODIFY und EV_VN_DELETE.

Laden und Speichern der Dokumente des Servers

>$IOLE-Anwendungen;SpeichernWenn ein Server Objekte an einen Container übergibt, so übernimmt der Container die Aufgabe des Speicherns der Objekte in Dateien und bei Bedarf das Zurücklesen. Wenn Ihr Server auch unabhängig laufen und seine eigenen Dokumente laden und speichern kann, sollte er auch von den Möglichkeiten der Verbunddateien Gebrauch machen, die in *TOleDocument* eingebunden sind. Eine Beschreibung von Verbunddateien finden Sie unter »Laden und Speichern von Verbunddokumenten«.

Da *TOleDocument* von *TStorageDocument* anstatt von *TFileDocument* abgeleitet ist, erzeugt es stets Verbunddateien. Wenn Sie *Read* oder *Write* aufrufen, stellt ObjektComponents automatisch einen Speicherbereich in Ihrer Datei für Ihre Dokumentdaten zur Verfügung.

;OLE- Ein Server sollte *TOleDocument::Write* in seiner Methode *Commit* und *TOleDocument::Read* in seiner Methode *Open* aufrufen.

```
// Deklaration der Dokumentklasse, abgeleitet von TOleDocument
class _DOCVIEWCLASS TMyDocument : public TOleDocument {
   // Deklarationen
}

// Implementierung der Dokumentklasse
```

```
bool TMyDocument::Commit(bool force) {
   TOleDocument::Write();   // Verbunddatei für die Verwendung durch
Commit vorbereiten
   . . .                    // Code zum Speichern anderer Dokumentdaten
}
bool TDrawDocument::Open(int, const char far* path) {
   TOleDocument::Read();    // Verbunddatei für die Verwendung durch Open
vorbereiten
   . . .                    // Code zum Speichern anderer Dokumentdaten
}
```

4. Erzeugen des Servers

Zum Erzeugen der Server-Anwendung binden Sie die OLE-Header ein und linken Sie sie mit den OLE-Bibliotheken.

Einbinden der OLE-Header

Die Header-Dateien für einen Server sind dieselben wie die für einen Container. Ein Server, der das Dok/View-Modell und ein MDI-Rahmenfenster benutzt, benötigt die folgenden Header-Dateien:

```
#include /oledoc.h     // ersetzt DOCVIEW.H
#include /oleview.h    // ersetzt DOCVIEW.H
#include /olemdifr.h   // ersetzt MDI.H
```

Bei einer SDI-Anwendung bindet man oleframe.h an Stelle von olemdifr.h ein.

Compilieren und Linken

OLE-Server, die ObjectComponents und ObjectWindows benutzen, erfordern das Speichermodell Large. Linken Sie sie mit OLE- und ObjektComponents-Bibliotheken.

Die integrierte Entwicklungsumgebung (IDE) wählt die passenden Erzeugungsoptionen aus, wenn Sie die OLE-Unterstützung anfordern. Um ein beliebiges ObjektComponents-Programm von der Kommandozeile aus zu erzeugen, schreiben Sie sich eine kurze Make-Datei, die die Datei OWLOCFMK.GEN (zu finden im Unterverzeichnis EXAMPLES) einbindet.

```
EXERES = MYPROGRAM
OBJEXE = winmain.obj myprogram.obj
!include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES und OBJEXE speichern die Namen der zu erstellenden Datei und die Namen der Objektdateien, aus denen sie aufgebaut wird. Die letzte Zeile sorgt für die Einbindung der Datei OWLOCFMK.GEN. Nennen Sie Ihre Datei MAKEFILE und geben Sie an der Eingabeaufforderung der Kommandozeile folgendes ein:

```
make MODEL=l
```

MAKE benutzt die Anweisungen in OCFMAKE.GEN und erzeugt eine neue, auf Ihr Projekt zugeschnittene Make-Datei mit dem Namen WIN16L*xx*.MAK.

Umwandeln einer ObjectWindows-Anwendung in einen OLE-Server

Zum Umwandeln eines ObjectWindows-Containers ohne Dok/View-Modell in einen OLE-Server sind einige Modifikationen erforderlich. Die folgende Liste beschreibt sie in Kurzfassung. Die anschließenden Abschnitte enthalten die Details für jede einzelne Modifikation.

1. Registrieren Sie den Server.
   - Erzeugen Sie ein Anwendungs-Dictionary-Objekt.
   - Erzeugen Sie die Registriertabellen für den Server.
   - Verbinden Sie die Dokumentregistrierstrukturen in einer verketteten Liste.
   - Erzeugen Sie das Registrar-Objekt und rufen Sie seine Methode *Run* auf.
2. Richten Sie das Client-Fenster ein.
3. Modifizieren Sie die Anwendungsklasse.
   - Leiten Sie sie von *TOcModule* ab, ebenso wie von *TApplication*.
   - Fügen Sie die Methode *CreateOleObjekt* hinzu.
4. Erzeugen Sie die Anwendung.

Die folgenden Abschnitte erläutern ausführlich jeden Schritt, der zur Umwandlung Ihres ObjectWindOws-Containers in eine OLE-Server erforderlich ist. Die Code-Fragmente stammen aus dem Beispiel OWLOCF2.CPP im Verzeichnis EXAMPLES/OWL/TUTORIAL/OLE. OWLOCF2 wandelt das Beispiel OWLOCF1 von einen Container in einen Server um.

1. Registrieren des Servers

Um den Server zu registrieren, beschreiben Sie ihn in Registriertabellen: eine Tabelle für die Anwendung und je eine Tabelle für jeden Dokumenttyp, den sie erzeugt. Die Dokumenttabellen werden als verkettete Liste gespeichert, das Registrar-Objekt verarbeitet die Informationen aus allen Tabellen. Das Registrar-Objekt benötigt außerdem ein Anwendungs-Dictionary-Objekt.

Erzeugen eines Anwendungs-Dictionarys

Ein Anwendungs-Dictionary verwaltet Informationen über den gerade aktiven Prozeß. Es ist besonders für DLLs nützlich. Wenn mehrere Prozes-

se eine DLL gemeinsam benutzen, so muß diese mehrere Kopien der globalen, statischen und dynamischen Variablen verwalten, die den aktuellen Zustand der DLL im jeweiligen Prozeß repräsentieren. Beispiel: Die DLL-Version von ObjectWindows enthält ein Dictionary, das es ermöglicht, das Objekt der Klasse *TApplication* abzurufen, das zu dem gerade aktiven Client-Prozeß gehört. Wenn Sie einen ausführbaren Server in einen DLL-Server umwandeln, so muß letzterer ein Dictionary der Objekte der Klasse *TApplication* verwalten, von denen jedes einen seiner Container-Clients repräsentiert.

Das Makro DEFINE_APP_DICTIONARY stellt einen einfachen und vereinheitlichten Weg zur Erzeugung des Dictionary-Objekts für jede Art von Anwendung bereit, ganz gleich, ob es ein Container, ein Server, eine DLL oder ein EXE-Programm ist. Fügen Sie dazu die folgende Anweisung bei Ihren Deklarationen der anderen statischen Variablen ein:

```
DEFINE_APP_DICTIONARY(AppDictionary);
```

Das Makro erzeugt einfach für jede zu der statischen Version der DLL-Bibliothek hinzugelinkte Anwendung eine Referenz auf das Anwendungs-Dictionary in ObjectWindows. Für DLL-Server, die die DLL-Version ObjectWindows benutzen, erzeugt es statt dessen eine Instanz der Klasse *TAppDictionary*.

Es ist wichtig, daß Sie Ihrem Verzeichnisobjekt den Namen *AppDictionary* geben, um die Vorteile von Factory-Templates wie etwa *TOleDocViewFactory* (wie später erklärt unter »Erzeugen eines Registrar-Objekts«) zu nutzen.

Erzeugen von Registriertabellen

Server implementieren OLE-Objekte, die jeder Container mit seinen eigenen Dokumenten verknüpfen oder darin einbetten kann. Unterschiedliche Server implementieren unterschiedliche Typen von Objekten. Jeder Typ von Objekt, den ein Server erzeugt, muß einen global eindeutigen Namen (GUID) aus 16 Zeichen und einen eindeutigen String-Bezeichner haben. Jeder Server muß diese Information zusammen mit anderen beschreibenden Informationen in die Registrierungsdatenbank des Systems, unter dem er läuft, schreiben. Das OLE-System liest die Registrierungen, um zu ermitteln, welche Objekte verfügbar sind, welche Fähigkeiten sie haben und wie die Anwendung aufgerufen werden muß, die Objekte eines bestimmten Typs erzeugt.

Ein Server stellt ObjektComponents die Registrierinformation zur Verfügung, indem er Makros zum Erzeugen der Registriertabellen benutzt: Eine Tabelle zur Beschreibung der Anwendung selbst und je eine Tabelle für jeden Typ von OLE-Objekten, die der Server erzeugt. Es folgt die Anwendungsregistriertabelle aus OWLOCF2.

```
REGISTRATION_FORMAT_BUFFER(100)

// Anwendungsregistriertabelle
BEGIN_REGISTRATION(AppReg)
   REGDATA(clsid,       "{B6B58B70-B9C3-101B-B3FF-86C8A0834EDE}")
   REGDATA(description,"Server für Skizzenblatt")

END_REGISTRATION
```

Die Registriermakros erzeugen eine Struktur aus mehreren Einträgen. Jeder Eintrag enthält einen Schlüssel, wie zum Beispiel *clsid* oder *description* und einen Wert, der diesem Schlüssel zugewiesen ist. Die Reihenfolge, in der die Schlüssel gespeichert sind, ist ohne Bedeutung. Im Beispiel ist *AppReg* der Name der Struktur, die die Informationen dieser Tabelle enthält.

Server, die mehrere Typen von Objekten erzeugen, müssen für jeden Dokumenttyp eine Dokumentregistriertabelle anlegen. (Was der Server als Dokument erzeugt, wird über das OLE-System als Objekt weitergegeben.) Wenn zum Beispiel eine Tabellenkalkulationsanwendung Tabellenkalkulations- und Geschäftsgrafikdateien erzeugt und wenn beide Arten von Dateien verknüpft oder eingebettet werden können, so registriert diese Anwendung zwei Dokumenttypen und erzeugt zwei Dokumentregistriertabellen.

Das Beispiel OWLOCF2 erzeugt einen Objekttyp, nämlich ein Skizzenblatt, deshalb benötigt es nur eine Dokumentregistriertabelle (hier gezeigt) zusätzlich zu der Anwendungsregistriertabelle.

```
// Dokumentregistriertabelle

BEGIN_REGISTRATION(DocReg)
   REGDATA(progid,      "Skizze.Dokument.3")
   REGDATA(description,"Skizzenblattdokument")
   REGDATA(debugger,    "tdw")
   REGDATA(debugprogid,"Skizze.Dokument.3.D")
   REGDATA(debugdesc,   "Skizzenblattdokument (Debuggen)")
   REGDATA(menuname,    "Skizzenblatt")
   REGDATA(insertable,  "")
   REGDATA(extension,   DocExt)
   REGDATA(docfilter,   "*."DocExt)
   REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt |
dtRegisterExt)
   REGDATA(verb0,       "&Edit")
   REGDATA(verb1,       "&Open")
   REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
   REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

Der Schlüssel *progid* ist ein Bezeichner für diesem Dokumententyp. Der String muß eindeutig sein, so daß das OLE-System die einzelnen Objekte

voneinander unterscheiden kann. Der Schlüssel *insertable* zeigt an, daß dieser Dokumententyp in Container eingesetzt werden kann. Die Schlüssel *description, menuname* und *verb* sind alle während OLE-Aktionen für den Benutzer sichtbar. Der Schlüssel *description* erscheint im Dialogfenster *Einfügen/Objekt*, wo dem Benutzer eine Liste der im System verfügbaren Objekte gezeigt wird. Der Schlüssel *menuname* wird für das Menü *Bearbeiten* des Containers benutzt, die Wahl des entsprechenden Menüpunkts führt zum Öffnen eines Popup-Menüs, in dem dann die Strings der Schlüssel *verb* erscheinen und die Wahl einer Aktionsart ermöglichen. Die restlichen Registriereinträge werden benutzt, wenn die Anwendung eine Datei öffnet oder die Windows Zwischenablage benutzt.

Tabelle 21.1. enthält eine Liste der Schlüssel, die ein Server registrieren sollte. Weitere Informationen über einzelne Schlüssel finden Sie in dem *ObjectWindows Referenzhandbuch*.

Schreiben Sie Ihre Registrierstrukturen dort in die Quelltextdatei, wo Sie Ihre von *TApplication* abgeleitete Klasse implementieren. Wenn Sie eine Registriertabelle per Kopieren und Ausschneiden und Einfügen aus anderen Programmen erzeugen, so vergewissern Sie sich, daß Sie mindestens die Einträge *progid* und *clsid* geändert haben, denn diese müssen Ihre Anwendung eindeutig identifizieren. (Verwenden Sie das Hilfsprogramm GUIDGEN.EXE zum Erzeugen von neuen 16-Zeichen-Werten für *clsid*.)

Erzeugen der Dokumentliste

Die Registriertabellen enthalten Informationen über Ihre Anwendung und deren Dokumente, aber sie sind statisch. Sie tun nichts mit diesen Informationen. Um die Informationen in einem System zu registrieren, muß eine Anwendung die Strukturen einem Objekt übergeben, das weiß, wie sie zu benutzen sind. Dieses Objekt ist das Registrar-Objekt, es schreibt alle nötigen Informationen in die Registrierungsdatenbank des Systems.

In einer Dok/View-Anwendung geht das Registrar-Objekt die Liste der Dokument-Templates durch, um die Dokumentregistrierstruktur jedes Dokuments zu finden. Eine nicht mit Dok/View arbeitende Anwendung besitzt keine Dokument-Templates, deshalb benutzt sie statt dessen *TRegLink*, um eine verkettete Liste aller ihrer Dokumentregistriertabellen zu erzeugen.

```
static TRegLink* RegLinkHead;
TRegLink scribbleLink(::DocReg, RegLinkHead);
```

RegLinkHead zeigt auf den ersten Knoten der verketteten Liste. *scribbleLink* ist ein Knoten in dieser Liste. Der Konstruktor *TRegLink* folgt *RegLinkHead* zum Ende der Liste und fügt den neuen Knoten an. Jeder Knoten enthält einen Zeiger auf eine Dokumentregistrierstruktur. In

OWLOCF2 enthält die Liste nur einen Knoten, da der Server nur Dokumente eines Typs erzeugt. Dieser Knoten zeigt auf *DocReg*.

OWLOCF2 deklariert *RegLinkHead* als statische Variable, da sie in mehreren Programmteilen verwendet wird, wie noch in den die folgenden Abschnitten erläutert wird.

Erzeugen des Registrar-Objekts

Das Registrar-Objekt registriert und startet die Anwendung. Seinem Konstruktor werden die Anwendungsregistrierstruktur und ein Zeiger auf die Liste der Dokumentregistrierstrukturen übergeben. In einer OLE-Anwendung hat das Registrar-Objekt den Typ *TOcRegistrar*. Deklarieren Sie am Anfang Ihrer Quelltextdatei eine globale Variable, die einen Zeiger auf das Registrar-Objekt aufnimmt:

```
static TPointer
  gistrar Registrar;
```

Das Template *TPointer* stellt sicher, daß die Instanz der Klasse *TOcRegistrar* gelöscht wird, wenn das Programm endet.

Hinweis

Geben Sie dieser Variablen den Namen *Registrar*, um die Vorteile zu nutzen, die das Factory Callback-Template im Konstruktor des Registrars bietet.

Erzeugen Sie im nächsten Schritt in *OwlMain* ein neues Objekt der Klasse *TOcRegistrar* und initialisieren Sie die globale Zeigervariable *Registrar*. Der Konstruktor von *TOcRegistrar* erwartet drei Parameter: die Anwendungsregistrierstruktur, die Factory Callback-Funktion der Komponente und den Kommandozeilen-String, mit dem die Anwendung aufgerufen wurde.

Ein optionaler vierter Parameter zeigt auf den Anfang der Liste der Dokumentregistrierstrukturen. In einer Dok/View-Anwendung zeigt dieser Parameter in der Voreinstellung auf die Liste der Dokument-Templates der Anwendung. Anwendungen, die Dok/View nicht benutzen, sollten einen Zeiger auf *TRegLink**, die Liste der Dokumentregistrierstrukturen übergeben.

```
int OwlMain(int /*argc*/, char* /*argv*/ []){
   try {
      // Erzeugen eines Registrar-Objekt zum Registrieren der Anwendung
      Registrar = new TOcRegistrar(::AppReg,    // Anwendung registration
structure
                  TOleFactory
  App(),    // Factory-Callback
                  TApplication::GetCmdLine(),     // Kommandozeile der
Anwendung
                  ::RegLinkHead);                 // Zeiger auf die
Dokumentregistrierstrukturen
```

```
   // Schalter "Nur registrieren" in der Kommandozeile?
   if (Registrar->IsOptionSet(amAnyRegOption))
      return 0;
   return Registrar->Run();                    // Ereignisschleife in
Factory Callback-Funktion
   }
   catch (xmsg& x) {
      ::MessageBox(0, x.why().c_str(), "Fehler in
Skizzenblock-Anwendung", MB_OK);
   }
   return -1;
}
```

TOleFactory ist eine Template-Klasse, die eine Factory Callback-Funktion erzeugt. Für eine ObjectWindows-Anwendung mit OLE-Fähigkeiten, die Dok/View nicht benutzt, heißt die Template-Klasse *TOleFactory*. Der Code im Factory Template geht davon aus, daß Sie ein Anwendungs-Dictionary mit dem Namen *AppDictionary* und einen Zeiger auf *TOcRegistrar* mit dem Namen *Registrar* definiert haben.

Wenn das Registrar-Objekt erzeugt wird, vergleicht es die Informationen in den Registriertabellen mit den Einträgen für diese Anwendung in der Registrierungsdatenbank des Systems und aktualisiert die Datenbank nötigenfalls. Die Methode *Run* bewirkt, daß das Registrar-Objekt die Factory Callback-Funktion aufruft, die unter anderem die Ereignisschleife der Anwendung anspringt.

Weitere Informationen dazu, was das Registrar-Objekt auf der Kommandozeile sucht, die Sie ihm übergeben, finden Sie unter »Verarbeiten der Kommandozeile«.

2. Einrichten des Client-Fensters

ObjektComponents-Anwendungen müssen ein separates Fenster für jedes Dokument haben. Das Dokumentfenster ist von *TOleFrame* abgeleitet und wird normalerweise so angelegt, daß es den Client-Bereich eines Rahmenfensters ausfüllt. Eine Erläuterung der Art und Weise, in der das Beispielprogramm OWLOCF ein Client-Fenster benutzt, finden Sie unter »Benutzen von Client-Fenstern«. Folgen sie den dortigen Anweisungen, um ein Client-Fenster hinzuzufügen, falls Ihre Anwendung nicht bereits eines besitzt.

Erzeugen von Helper-Objekten für ein Dokument

Jedes neue Dokument, das sie öffnen, benötigt zwei Helper-Objekte von ObjektComponents, *TOcDocument* und *TOcView*. Da Sie für jedes Dokument ein Client-Fenster erzeugen, ist der Konstruktor des Fensters ein geeigneter Ort für die Erzeugung der Helper-Objekten. Die Funktion *TOleWindow::CreateOcView* erzeugt beide gleichzeitig.

In OWLOCF2 ist *TScribbleWindow* das Client-Fenster. Hier folgen die Deklarationen für die Klasse und ihren Konstruktor:

```
class TScribbleWindow : public TOleWindow {
  public:
    TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData& fileData);
    TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData& fileData,
TRegLink* link);
```

Der zweite Konstruktor ist neu. Er ist nützlich, wenn ObjektComponents ihm einen Zeiger auf die von ihm für einen Ihrer Dokumenttypen bereitgestellten Registrierinformationen übergibt und Sie auffordert, ein Dokument dieses Typs zu erzeugen. Hier folgt die Implementierung des neuen Konstruktors:

```
TScribbleWindow::TScribbleWindow(TWindow* parent, TOpenSaveDialog::TData&
fileData,     TRegLink* link)
:
  TOleWindow(parent, 0),
  FileData(fileData)
{
  . . .
  // Erzeugen eines OcDocument-Objekts, das die OLE-Informationen
aufnimmt, die wir erzeugen;
  // Erzeugen eines OcRemView-Objekts, das OLE-Dienste bereitstellt
  CreateOcView(link, true, 0);
}
```

Dem Konstruktor wird ein Zeiger *TRegLink* übergeben und reicht ihn an *CreateOcView* weiter. Der Zeiger zeigt auf die Dokumentregistrierfunktion für den Typ des erzeugten Dokuments. ObjektComponents übergibt den Zeiger an diesen Konstruktor, Sie müssen sich nicht selbst darum kümmern.

Die Übergabe des Werts <Wahr> an *CreateOcView* veranlaßt die Funktion, ein Helper-Objekt vom Typ *TOcRemView* an Stelle von *TOcView* zu erzeugen. Das Remote View-Objekt stellt ein OLE-Objekt innerhalb des Fensters eines Containers grafisch dar. Wenn ein Server gestartet wird, um einen Client mit einem OLE-Objekt zu unterstützen, sollte er eine Remote View erzeugen.

Wenn Ihre Anwendung mehr als einen Dokumenttyp unterstützt, können Sie sich entscheiden, für jeden Typ eine andere von *TOleWindow* abgeleitete Klasse zu verwenden. Sie müssen dann den zusätzlichen Konstruktor für jede Klasse bereitstellen. Eine andere Möglichkeit ist, eine einzige, von *TOleWindow* abgeleitete Klasse zu verwenden, die in Abhängigkeit von dem ihr übergebenen Zeiger auf *TRegList* unterschiedliche Verhaltensweisen hat.

3. Modifizieren der Anwendungsklasse

ObjektComponents fordert, daß die Klasse, die Sie von *TApplication* ableiten, auch von *TOcModule* erbt, wie es unter »Ableiten des Anwendungsobjekts von TOcModule« erläutert wird. Außerdem muß das Anwendungsobjekt die Methode *CreateOleObjekt* mit folgendem Funktionskopf erzeugen:

```
TUnknown* CreateOleObjekt(uint32 options, TRegList* link);
```

Der Zweck der Funktion besteht darin, ein Server-Dokument zum Verknüpfen und Einbetten zu erzeugen. Der Server muß ein Client-Fenster erzeugen und einen Zeiger des Typs *TOcRemView** zurückgeben. OWLOCF2 deklariert diese Funktion wie folgt:

```
class TScribbleApp : public TApplication, public TOcModule {
  public:
    TScribbleApp();
    TUnknown* CreateOleObjekt(uint32 options, TRegLink* link);
```

Implementiert wird die Funktion in OWLOCF2 in folgender Form:

```
TUnknown*
TScribbleApp::CreateOleObjekt(uint32 options, TRegLink* link)
{
  if (!link)  // Factory-Funktion erzeugt nur eine Anwendung, View nicht
gefordert
    link = &scribbleLink;       // View für diese Anwendung erforderlich
  TOleFrame* olefr = TYPESAFE_DOWNCAST(GetMainWindow(), TOleFrame);
  CHECK(olefr);
  FileData.FileName[0] = 0;
  TScribbleWindow* client = new TScribbleWindow(
          olefr->GetRemViewBucket(), FileData, link);
  client->Create();
  return client->GetOcRemView();
}
```

ObjectWindows benutzt die Methode *CreateOleObjekt*, um Ihre Anwendung zu informieren, wann das OLE-System den Server zum Erzeugen eines Objekts benötigt. Der Parameter *TRegList* * liefert die Information darüber, welches Objekt erzeugt werden soll.

Funktion der Dokumentenliste TRegLink

Dieser Abschnitt erläutert die Beziehung zwischen der Dokumentregistrierstruktur, der Dokumentliste und der Methode *CreateOleObjekt*. Ein Server erzeugt eine Dokumentregistriertabelle für jeden Objekttyp, den er als Server unterstützt. (Die Variable, die die Dokumentregistrierinformation speichert, wird normalerweise *DocReg* genannt.) Die Registrierstruktur wird dann an einen Konstruktor vom Typ *TRegLink* übergeben, der die Struktur so an eine verkettete Liste anhängt, daß alle vorhandenen Dokumenttypen registriert werden können.

Das OLE-System zeigt den Wert von *description* für jedes Objekt im Dialogfeld *Einfügen* immer dann an, wenn der Bediener ein Objekt einfügen will. (Das OLE-System zeigt außerdem die Strings *description* für alle anderen verfügbaren Typen von Server-Dokumenten an.) Wenn der Bediener eines Ihrer Objekte in eine Container-Anwendung einfügen will, startet das OLE-System Ihren Server und setzt den Schalter *-Embedding* in die Kommandozeile. Nachdem der Server geladen ist, ruft ObjektComponents Ihre Methode *CreateOleObjekt* auf und übergibt ihr dabei einen Zeiger auf die verkettete Registrierliste, die zum Registrieren des angeforderten Dokumenttyps benutzt wurde. Der Zeiger *TRegList* ermöglicht es Ihnen, festzustellen, welchen Objekttyp der Bediener gewählt hat. Dies ist hauptsächlich für solche Server von Interesse, die mehr als einen Objekttyp registriert haben.

Der folgende Code illustriert eine mögliche Implementierung der Methode *CreateOleObjekt* für eine Anwendung, die als Server mehr als einen Objekttyp bereitstellt:

```
//
// Erzeugen eines passenden Client-Fensters und Rückgabe seines Zeigers
TOcRemView
//
TUnknown*
TServerApp::CreateOleObjekt(uint32 options, TRegList* link)
{
  if (link == &chartLink) {
    // TOleChartWindow erzeugen
    // und charWindow-GetOcRemView() zurückgeben;
  }

  if (tpl == &worksheetLink) {
    // TOleWorksheetWindow erzeugen
    // und worksheetWindow-GetOcRemView() zurückgeben;
  }
  return 0;
}
```

4. Erzeugen eines Servers

Um eine Server-Anwendung zu erzeugen, binden Sie die OLE-Include-Dateien ein, und linken Sie mit dem OLE-Bibliotheken.

Einbinden der OLE-Header-Dateien

ObjectWindows stellt OLE-fähige Klassen, Strukturen, Makros und Symbole in verschiedenen Include-Dateien bereit. Die folgende Liste zeigt die für einen OLE-Container, der ein SDI-Rahmenfenster benutzt, benötigten Include-Dateien:

```
#include \oleframe.h
#include \olewindo.h
#include \ocstorag.h
```

Eine MDI-Anwendung benötigt die Include-Datei olemdifr.h an Stelle von oleframe.h.

Umwandeln einer C++ Anwendung in einen OLE-Server

Wenn Sie ein neues Programm schreiben wollen, sollten Sie es erwägen, ObjectWindows zu benutzen, um einige Arbeiten zu sparen. Die ObjectWindows-Bibliothek enthält integrierten Code, der einige allen ObjectComponents-Programmen gemeinsame Aufgaben automatisch ausführt. Programme, die ObjectWindows nicht benutzen, müssen diese Dinge selbst erledigen.

Die folgende Liste beschreibt in Kurzform, was Sie tun müssen, um eine C++ Anwendung in einen ObjectComponents-Server umzuwandeln. Viele Punkte dieser Liste finden sich auch in der Liste der Änderungen zum Erzeugen eines Containers (siehe dort). Falls Sie Ihre C++ Anwendung bereits in einen Container umgewandelt haben, ist der größte Teil der Arbeit zur Umwandlung in einen Server schon erledigt. Die wichtigsten Unterschiede betreffen die Registriertabellen und die Factory Callback-Funktion.

Die auf die Liste folgenden Abschnitte beschreiben jeden Schritt im Detail.

1. Erzeugen Sie Objekt zur Zuweisung von Speicher in *WinMain*.
2. Registrieren Sie die Anwendung.
   - Erzeugen Sie die Registriertabellen.
   - Erzeugen Sie eine Dokumentenliste.
   - Erzeugen Sie ein Registrar-Objekt und rufen Sie seine Funktion *Run* auf.
   - Schreiben Sie eine Factory Callback-Funktion, die auf Anforderung die Anwendung oder deren Dokumente erstellt.
3. Erzeugen Sie ein View-Fenster zur Anzeige eines geöffneten Dokuments.
   - Erzeugen und Löschen Sie das View-Fenster, bzw. verändern Sie seine Größe zusammen mit dem Hauptfenster.
   - Erzeugen Sie Hilfsobjekte für ein neues Server-Dokument.
   - Lassen Sie das View-Fenster die Botschaft WM_OCEVENT behandeln.
   - Schreiben Sie Behandlungsroutinen für bestimmte ObjectComponents-View-Ereignisse.
   - Stellen Sie das Dokument des Servers im View-Fenster grafisch dar.
4. Lassen Sie das Hauptfenster OLE-Befehle und Ereignisse behandeln.
   - Rufen Sie beim Erzeugen des Hauptfensters *::SetupWindow* auf.

- Lassen Sie das Hauptfenster die Botschaft WM_OCEVENT behandeln.
- Schreiben Sie Behandlungsroutinen für bestimmte ObjectComponents-Anwendungsereignisse.

5. Erstellen Sie die Anwendung.

- Binden Sie die Header-Dateien von ObjectComponents ein.
- Linken Sie dann die OLE- und ObjectComponents-Bibliotheken hinzu.

Die folgenden Abschnitte illustrieren jeden der Schritte mit Beispielen aus den im Verzeichnis EXAMPLES/OCF/CPPOCF enthaltenen Programmen. Die Quelltextdateien mit dem Namen CPPOCF0 enthalten eine Windows Anwendung, die OLE nicht unterstützt. CPPOCF1 wandelt das erste Programm in einen OLE-Container um. CPPOCF2 fügt Server-Fähigkeiten hinzu. Die hier betrachteten Code-Beispiele stammen aus CPPOCF2. Der Server von CPPOCF2 erzeugt eine einfache Zeitanzeige für Container zum Einbetten. Der angezeigte Zeitwert wird jede Sekunde erhöht.

Dieses Kapitel beschreibt nicht alle Änderungen von CPPOCF1 nach CPPOCF2, sondern nur die, die mit OLE zusammenhängen.

1. Erzeugen eines Speicherreservierungsobjekts

Bei einem Server wird die folgende Zeile am Beginn seiner Funktion *WinMain* hinzugefügt:

```
TOleAllocator allocator(0);      // mit standardmäßiger
Speicherreservierung arbeiten
```

Der Konstruktor des Reservierungsobjekts initialisiert die OLE-Bibliotheken, der Destruktor gibt sie frei, wenn das Objekt den Gültigkeitsbereich verläßt. Die Übergabe des Werts 0 teilt dem Konstruktor mit, daß das OLE-System seine standardmäßigen Speicherverwaltungsfunktionen bei jeder Speicherreservierung für diese Anwendung benutzen soll.

2. Registrieren der Anwendung

CPPOCF2 unterstützt grundlegende Server-Funktionen. Es registriert Informationen über sich selbst und seinen Dokumenttyp, und es erzeugt auf Anforderung ein Objekt zur Einbettung in andere Anwendungen.

Erzeugen der Registriertabellen

Ein Server benutzt die Registriermakros zum Erzeugen von Registriertabellen, die die Anwendung und die von ihr erzeugten Dokumente beschreiben. Die erste Tabelle beschreibt den Server selbst:

```
REGISTRATION_FORMAT_BUFFER(100)
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,        "{BD5E4A81-A4EF-101B-B31B-0694B5E75735}")
  REGDATA(description,  "Beispiel eines C-Servers")
END_REGISTRATION
```

Die Registriermakros erzeugen eine Struktur des Typs *TRegList*. Die Struktur wird in einer Variablen mit dem Namen *AppReg* gespeichert. Jedes Element der Struktur enthält einen Schlüssel, zum Beispiel *clsid* oder *description,* und einen dem Schlüssel zugewiesenen Wert. Intern findet ObjectComponents diese Werte durch Suchen nach dem Schlüssel. Die Reihenfolge, in der die Schlüssel gespeichert werden, ist ohne Bedeutung.

Der Server erzeugt eine zweite Registriertabelle, die die Typen der von ihm erzeugten Dokumente beschreibt. Wenn beispielsweise eine Tabellenkalkulationsanwendung, Tabellenkalkulations- und Geschäftsgrafikdateien erzeugt und wenn beide Arten von Dateien verknüpft oder eingebettet werden können, so registriert die Tabelle zwei Dokumenttypen. CPPOCF2 erzeugt nur Dokumente eines Typs, eine Zeitanzeige. Die Registrierstruktur für diesen Dokumenttyp wird in einer Variablen mit dem Namen *DocReg* gespeichert, wie der folgende Code zeigt.

```
BEGIN_REGISTRATION(DocReg)
  REGDATA(description,  "Beispiel für ein C-Server-Dokument")
  REGDATA(progid,       APPSTRING".Document.1")
  REGDATA(menuname,     "CServer")
  REGDATA(insertable,   "")
  REGDATA(verb0,        "&Edit")
  REGDATA(verb1,        "&Open")
  REGDATA(extension,    "scd")
  REGDATA(docfilter,    "*.scd")
  REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
  REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
  REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

Der Schlüssel *progid* ist ein Bezeichner für diesem Dokumententyp. Der Schlüssel *insertable* zeigt an, daß dieser Dokumententyp in Container eingesetzt werden kann. Die Schlüssel *description, menuname* und *verb* sind alle während der OLE-Aktionen für den Benutzer sichtbar. Der Schlüssel *description* erscheint im Dialogfenster *Objekt einfügen,* wo dem Benutzer eine Liste der im System verfügbaren Objekte gezeigt wird. Der Schlüssel *menuname* wird für das Menü *Bearbeiten* des Containers benutzt, die Wahl des entsprechenden Menüpunkts führt zum Öffnen eines Popup-Menüs, in dem dann die Strings der Schlüssel *verb* erscheinen und die Wahl einer Aktionsart ermöglichen.

Die restlichen Registrierungseinträge werden benutzt, wenn die Anwendung eine Datei öffnet oder die Windows Zwischenablage benutzt. Tabelle 21.1 stellt eine Liste der Schlüssel, die ein Server registrieren sollte, dar. Weitere Informationen über einzelne Schlüssel sind im *ObjectWindows Referenzhandbuch* zu finden.

Erzeugen der Dokumentliste

Die Registriertabellen enthalten Informationen über Ihre Anwendung und deren Dokumente, aber sie sind statisch. Sie tun nichts mit diesen Informationen. Um die Informationen in einem System zu registrieren, muß eine Anwendung die Strukturen einem Objekt übergeben, das weiß, wie sie zu benutzen sind. Dieses Objekt ist das Registrar-Objekt, es schreibt alle nötigen Informationen in die Registrierdatenbank des Systems.

Um Server mit mehreren Dokumenttypen möglich zu machen, speichert das Registrar-Objekt einen Zeiger auf eine verkettete Liste aller Dokumentregistrierstrukturen der Anwendung. Jeder Knoten in der Liste ist ein Objekt der Klasse *TRegLink*. Jeder Knoten enthält einen Zeiger auf eine der Dokumentregistrierstrukturen und einen weiteren Zeiger auf den nächsten Knoten.

```
TRegLink *RegLinkHead = 0;
TRegLink  regDoc(DocReg, RegLinkHead);
```

RegLinkHead zeigt auf den ersten Knoten der verketteten Liste. *regDoc* ist ein Knoten in dieser Liste. Der Konstruktor *TRegLink* folgt *RegLinkHead* zum Ende der Liste und fügt den neuen Knoten an. Jeder Knoten enthält einen Zeiger auf eine Dokumentregistrierstruktur. In CPPOCF2 enthält die Liste nur einen Knoten, da der Server nur Dokumente eines Typs erzeugt. Dieser Knoten zeigt auf *DocReg*.

CPPOCF2 deklariert *RegLinkHead* als statische Variable, da auf sie von mehreren Programmteilen aus zugegriffen wird; dies wird noch in den folgenden Abschnitten erläutert werden.

Erzeugen des Registrar-Objekts

Das Registrar-Objekt schreibt Informationen über die Anwendung in die Registrierdatenbank des Systems, verarbeitet OLE-bezogene Schalter in der Kommandozeile der Anwendung und teilt dem OLE-System mit, daß der Server läuft. CPPOCF2 deklariert einen statischen Zeiger für das Registrar-Objekt:

```
TOcRegistrar* OcRegistrar = 0;
```

Erzeugen Sie Ihr Registrar-Objekt bei der Initialisierung der Anwendung in *WinMain*. Statt in eine Ereignisschleife zu gehen, rufen sie dann die Methode *Run* des Registrar-Objekts auf. Wenn *Run* davon zurückkehrt, wird die Anwendung geschlossen. Löschen Sie das Registrar-Objekt vor dem Programmende. Die Funktion *WinMain* in CPPOCF2 zeigt diese Schritte:

```
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, char far*
lpCmdLine, int nCmdShow)
{
  HInstance = hInstance;
  try {
    TOleAllocator allocator(0);     // mit standardmäßiger
```

```
Speicherreservierung arbeiten

    string cmdLine(lpCmdLine);      // Kommandozeile der Anwendung in
einen C++ String einsetzen

    // Registrar-Objekt erzeugen
    OcRegistrar = new TOcRegistrar(::AppReg,  // Anwendungsregistrier-
                                              // struktur
          ComponentFactory,                   // Factory-Callback-Funktion
          cmdLine,                            // Kommandozeile der
                                              // Anwendung
          ::RegLinkHead);                     // Liste der Dokument-
                                              // registrierstrukturen

    . . .    // hier Anweisungen für die Initialisierung der Anwendung

    if (OcRegistrar->IsOptionSet(amEmbedding))
      nCmdShow = SW_HIDE;

    . . .    // hier Anweisungen für die Initialisierung der Instanz

    OcRegistrar->Run();               // Factory-Funktion aufrufen,
                                      // veranlassen, daß sich die
                                      // Anwendung selbst erzeugt und
                                      // startet
    delete OcRegistrar;
  }
  catch (xmsg& x) {
    MessageBox(GetFocus(), x.why().c_str(), "Fehler aufgetreten", MB_OK);
  }
  return 0;
}
```

Dem Konstruktor *TOcRegistrar* werden vier Parameter übergeben:

::AppReg ist die Registrierstruktur der Anwendung, sie ist bereits durch die Registriermakros erzeugt worden (siehe den Abschnitt »Erzeugen der Registriertabellen«).

ComponentFactory ist eine Callback-Funktion, die im nächsten Abschnitt beschrieben wird. Sie ist zuständig für die Erzeugung aller benötigter OLE-Komponenten der Anwendung, einschließlich der Anwendung selbst. Außerdem enthält die Callback-Funktion die Ereignisschleife der Anwendung.

cmdLine ist ein Objekt vom Typ *string*, das die Kommandozeile der Anwendung aufnimmt. Das Registrar-Objekt durchsucht die Kommandozeile nach OLE-bezogenen Schaltern wie zum Beispiel *-Automation* oder *-Embedding* und setzt die internen Flags des Ablaufmodus dementsprechend.

::RegLinkHead zeigt auf die verkettete Liste der Dokumentregistrierstrukturen. (Siehe den Abschnitt »Erzeugen der Dokumentliste«.)

Während seiner Initialisierung prüft der Server, ob er vom OLE-System oder direkt vom Benutzer aufgerufen wurde. Das OLE-System startet den Server, wenn der Benutzer ein verknüpftes oder eingebettetes Objekt aktiviert, das der Server erzeugt hat. Das OLE-System setzt den Schalter *-Embedding* in die Kommandozeile des Servers, um anzuzeigen, daß der Server nur zur Unterstützung eines Clients und nicht als unabhängige Anwendung läuft. Wenn das Registrar-Objekt den Schalter *-Embedding* in der Kommandozeile findet, so setzt es ein internes Flag. Der Server prüft dieses Flag durch Aufruf von *IsOptionSet*. Falls das OLE-System die Anwendung gestartet hat, schreibt der Server nur in das Fenster des Containers, er muß dann kein eigenes Fenster anzeigen. CPPOCF2 setzt *nCmdShow* auf den Wert SW_HIDE, um zu verhindern, daß der nachfolgende Initialisierungs-Code das Hauptfenster anzeigt.

nCmdShow ist der Parameter, den Windows an *WinMain* übergibt und der den anfänglichen Status des Hauptfensters der Anwendung anzeigt. Eine sinnvoll programmierte Windows Anwendung übergibt den Wert an *ShowWindow*, und zwar unmittelbar nach dem Erzeugen des Hauptfensters.

Die Funktion *TOcRegistrar::Run* veranlaßt das Registrar-Objekt, die Callback-Funktion der Anwendung aufzurufen. In diesem Fall führt die Callback-Funktion die Ereignisschleife der Anwendung aus, damit läuft dann die Anwendung.

Schreiben der Factory Callback-Funktion

Die Factory Callback-Funktion wird von Ihnen geschrieben und dem Registrar-Objekt übergeben. Wenn es an der Zeit ist, daß die Anwendung laufen muß, oder wenn ein Container versucht, eins der Server-Objekte einzufügen, ruft ObjectComponents die Callback-Funktion auf.

Die Factory Callback-Funktion entscheidet, was zu tun ist, indem sie die erhaltenen Parameter liest und die Flags des Ablaufmodus abfragt, die das Registrar-Objekt gesetzt hat. Die Callback-Funktion hat den Namensvorsatz »Factory (Hersteller)«, weil sie auf Anforderung OLE-Komponenten-Objekte erzeugt.

Die Forderung, daß jede ObjectComponents-Anwendung eine Factory Callback-Funktion bereitstellen muß, führt zur Vereinheitlichung des Verfahrens der Erzeugung von Objekten. Normalerweise variiert dieses Verfahren in Abhängigkeit davon, ob die Anwendung ein Container oder ein Server ist, ob sie automatisiert ist, ob sie als DLL- oder als EXE-Programm läuft, und ob sie vom Benutzer direkt oder vom OLE-System aufgerufen wurde. Die Factory Callback-Funktion ermöglicht es, die Anwendung zu verändern und in einer Vielzahl von Art und Weisen laufen zu lassen, ohne neuen Code zu schreiben. Für weitere Informationen über

Factory-Callbacks siehe unter »Factory-Templates« im *ObjectWindows Referenzhandbuch*.

Ein Satz von Factory Template-Klassen wie etwa *TOleFactory* und *TOleAutoFactory* macht es leicht, Factory-Funktionen für ObjectWindows Programme zu erzeugen, in einem reinen C++ Programm müssen Sie jedoch die Factory-Funktion selbst schreiben.

Sie können einer Factory-Callback-Funktion jeden gewünschten Namen geben, aber sie muß folgendem Prototyp entsprechen:

```
IUnknown* ComponentFactory(IUnknown* outer, uint32 options, uint32 id);
```

outer wird benutzt bei der Aggregation von OLE-Objekten, damit sie als eine einzige Einheit funktionieren. Der Rückgabewert der Factory-Funktion wird ebenso für die Aggregation benutzt. Da Container nicht mit Aggregation arbeiten, ignoriert CPPOCF1 *outer* und gibt 0 zurück.

options enthält die Einzelbit-Flags, die den Ablaufmodus der Anwendung anzeigen. Das Registrar-Objekt setzt die Flags, wenn es die Schalter der Kommandozeile verarbeitet und bevor es die Factory Callback-Funktion aufruft. Die Factory-Funktion prüft die Flags, um festzustellen, was sie tun soll. Die möglichen Werte der Flags sind im Aufzählungstyp *TOcAppMode* definiert, sie haben Namen wie *amRun* und *amShutdown*.

id ist ein Bezeichner, der der Factory-Funktion mitteilt, von welchem Typ das zu erzeugende Objekt ist.

Die Eingangsparameter der Factory-Funktion können diese zur Ausführung einer von drei Aktionen veranlassen:

- Initialisieren der Anwendung. Wenn die Factory-Funktion das erste Mal läuft, erzeugt sie ein Objekt der Klasse *TOcModule*. *TOcModule* verbindet die Anwendung mit dem OLE-System durch Erzeugen eines Verbinderobjekts der Klasse *TOcApp*. Außerdem behandelt die Factory-Funktion in dieser Phase die Aggregation.
- Starten der Anwendung. Wenn das Flag *amRun* gesetzt ist, so geht die Factory-Funktion in die Ereignisschleife. Wenn der Server als DLL realisiert ist, so setzt das Registrar-Objekt beim Laden des Servers durch das OLE-System nicht das Flag *amRun*, der Server sollte dann nicht seine eigene Ereignisschleife aktivieren.
- Erzeugen eines Objekts. Der Parameter *id* informiert die Factory-Funktion darüber, von welchem Typ das zu erzeugende Objekt ist. Da CPPOCF2 nur einen Typ von Objekten erzeugt, wird dort nur geprüft, ob *id* größer als 0 ist. In Anwendungen, die mehrere Dokument-Templates registrieren, zeigt *id* auf das Template für das angeforderte Objekt.

Die Factory-Callback-Funktion in CPPOCF2 greift auf vier globale Variablen zu. Eine davon ist *OcRegistrar* und wurde weiter oben beschrieben. Eine andere ist *OcApp*.

```
TOcRegistrar* OcRegistrar = 0;
TOcApp*       OcApp       = 0;
```

TOcApp ist das Verbinderobjekt, das die OLE-Schnittstellen auf Anforderung der Anwendung implementiert. Eine der Aufgaben der Factory-Funktion ist es, das Verbinderobjekt zu erzeugen, wenn die Anwendung startet, und es zu löschen, wenn die Anwendung geschlossen wird.

Die anderen beiden globalen Variablen, *OcDoc* und *OcRemView*, werden im nächsten Abschnitt beschrieben.

Hier folgt die Factory-Callback-Funktion aus CPPOCF2:

```
IUnknown*
ComponentFactory(IUnknown* outer, uint32 options, uint32 id)
{
  IUnknown* ifc = 0;

  // Anwendung sarten oder schließen
  if (!OcApp) {
    if (options & amShutdown)  // keine Anwendung zu schließen!
      return 0;
    OcRegistrar->CreateOcApp(options, OcApp);
  } else if (options & amShutdown) {
    DestroyWindow(HwndMain);
    return 0;
  }

  // Aggregation, wenn ein outer-Zeiger übergeben wurde
  if (id == 0)
    OcApp->SetOuter(outer);

  // Ereignisschleife aktivieren, wenn das Flag amRun gesetzt ist
  if (options & amRun) {
    if ((options & amEmbedding) == 0) {
      HwndView = CreateViewWindow(HwndMain);
    }
    MSG msg;
    // standardmäßige Windows Ereignisschleife
    while (GetMessage(&msg, 0, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }

  // Dokument erzeugen, falls der Parameter id nicht 0 ist
  if (id) {
```

```
    OcDoc     = new TOcDocument(*OcApp);
    HwndView  = CreateViewWindow(HwndMain);
    OcRemView = new TOcRemView(*OcDoc, &DocReg);
    if (IsWindow(HwndView))
      OcRemView->SetupWindow(HwndView);
    ifc = OcRemView->SetOuter(outer);
  }
  return ifc;
}
```

Der Parameter *outer* der Factory-Funktion ist 0, so lange nicht irgendein anderes Objekt mit dem neu erzeugten Objekt ein Aggregat bildet. Objekte in einem Aggregat sind Komponenten, die zusammen als einzige Einheit agieren. Objekte können zur Laufzeit Aggregate bilden; Sie benötigen keinen Zugriff auf den Quelltext eines Objekts, um mit ihm eine Aggregation zu bilden. ObjectComponents unterstützt Aggregation durch Übergabe von *outer* an die Factory-Funktion der Anwendung. Wenn der Zeiger *outer* ungleich 0 ist, so zeigt er auf die Schnittstelle *IUnknown* eines anderen Objekts, welches sich das neu erzeugte Objekt unterordnen will. Um Aggregation zuzulassen, ruft die Factory-Funktion die Methode *SetOuter* für das Objekt auf, das sie gerade erzeugt, also entweder *TOcApp* oder *TOcRemView*. *SetOuter* gibt einen Zeiger auf die eigene Schnittstelle *IUnknown* des Objekts zurück. Die Factory-Funktion sollte denselben Zeiger zurückgeben.

Anmerkung

TOcApp::SetOuter wird nur aufgerufen, wenn eine Anwendung sich selbst automatisiert. CPPOCF2 enthält jedenfalls den Aufruf für den Fall, daß die Anwendung später in einen Automatisierungs-Server umgewandelt wird.

3. Erzeugen eines View-Fensters

ObjectComponents stellt eine Forderung an die Implementierung eines Servers: Das Server-Dokument muß sein eigenes Fenster haben, getrennt vom Hauptfenster der Anwendung. Für eine klare Unterscheidung nennen wir das Hauptfenster das Rahmenfenster, weil es von der Fensterart WS_THICKFRAME ist und einen sichtbaren Rahmen auf dem Bildschirm zeigt. Das zweite Fenster hat keinen sichtbaren Rand. Wir nennen es das View-Fenster, weil es der Ort ist, wo die Anwendung ihre Daten anzeigt. Das View-Fenster füllt den Client-Bereich des Rahmenfensters exakt aus, so daß das Rahmenfenster aus der Sicht des Benutzers das einzige Fenster ist. ObjectComponents benötigt das View-Fenster, weil es manche Ereignisbotschaften an die Anwendung und andere an den View senden können will.

In einer SDI-Anwendung wie dem Beispielprogramm CPPOCF2 steuert das Rahmenfenster das View-Fenster. Wenn das Rahmenfenster die Botschaft WM_SIZE empfängt, so verschiebt es das View-Fenster so, daß es

immer exakt im Client-Bereich des Rahmenfensters bleibt. Beim Empfang der Botschaft WM_CLOSE löscht es das View-Fenster und sich selbst.

In einer MDI-Anwendung erzeugt jedes untergeordnete Fenster sein eigenes View-Fenster. Das untergeordnete Fenster tut das, was das SDI-Rahmenfenster tut: Es erzeugt und verwaltet ein View-Fenster für das Dokument, das es anzeigt.

Erzeugen, Ändern der Größe und Löschen des View-Fensters

Vor der Erzeugung des View-Fensters muß die Anwendung erst eine Klasse für das Fenster registrieren. CPPOCF2 registriert beide Klassen in *InitAnwendung*.

CPPOCF2 erzeugt das View-Fenster in seiner Factory-Funktion, weil diese zuständig für das Erzeugen neuer Dokumente auf Anforderung ist. Wie Sie sehen werden, verbindet der Code für das View-Fenster das neue Dokument mit dem OLE-System, indem einige Helper-Objekte von ObjectComponents erzeugt werden. Die Factory-Funktion ruft diese Funktion auf, um das View-Fenster zu erzeugen:

```
HWND CreateViewWindow(HWND hwndParent)
{
  HWND hwnd = CreateWindow(VIEWCLASSNAME, "",
    WS_CHILD | WS_CLIPCHILDREN | WS_CLIPSIBLINGS | WS_VISIBLE | WS_BORDER,
    10, 10, 300, 300,
    hwndParent, (HMENU)1, HInstance, 0);
  return hwnd;
}
```

CPPOCF2 verändert die Größe des View-Fensters oder löscht es, wenn das Rahmenfenster die Botschaft WM_SIZE bzw. WM_CLOSE empfängt.

```
void
MainWnd_OnSize(HWND hwnd, UINT /*state*/, int /*cx*/, int /*cy*/)
{
  if (IsWindow(HwndView)) {
    TRect rect;
    GetClientRect(hwnd, &rect);
    MoveWindow(HwndView, rect.left, rect.top, rect.right, rect.bottom,
true);
  }
}

void
MainWnd_OnClose(HWND hwnd)
{
  if (IsWindow(HwndView))
    DestroyWindow(HwndView);
  DestroyWindow(hwnd);
}
```

Das View-Fenster füllt den Client-Bereich des Rahmenfensters stets exakt aus. Wenn der Benutzer Dokumente öffnet und schließt oder Objekte einbettet, so werden die Änderungen im View-Fenster sichtbar.

Erzeugen eines neuen Server-Dokuments

Für jedes geöffnete Dokument muß der Server zwei Helper-Objekte erzeugen, *TOcDocument* und *TOcRemView*. Das Helper-Objekt des Dokuments verwaltet die in das Dokument eingefügte Sammlung von Objekten. (Es ist möglich, Objekte in andere Objekte einzubetten.) Das Helper-Objekt des Views verbindet das Dokument mit dem OLE-System. Genauer gesagt, es implementiert Schnittstellen, die das OLE-System aufrufen kann, um mit dem Dokument zu kommunizieren. Wenn das OLE-System dem View-Objekt mitteilt, daß etwas Beachtenswertes geschehen ist, so sendet das View-Objekt eine Botschaft an das View-Fenster. (Die folgenden zwei Abschnitte zeigen die Behandlung dieser Botschaften.)

Der Server des Beispiels CPPOCF2 erzeugt die beiden Helper-Objekte in seiner Factory-Funktion, wenn er aufgefordert wird, eine neues Dokument zu erzeugen.

```
OcDoc     = new TOcDocument(*OcApp);
HwndView  = CreateViewWindow(HwndMain);
OcRemView = new TOcRemView(*OcDoc, &DocReg);
if (IsWindow(HwndView))
  OcRemView->SetupWindow(HwndView);
```

Die Methode *SetupWindow* teilt dem *TOcRemView* Objekt mit, wohin Ereignisbotschaften zu senden sind. Im Beispiel sendet es Botschaften an *HwndView*, also an das View-Fenster. Das View-Fenster empfängt nun Botschaften des Typs WM_OCEVENT.

TOcDoc, *TOcRemView* und ein View-Fenster sind die Strukturen, die ein Server für jedes neue Dokument benötigt. Ebendiese Objekte werden gelöscht oder freigegeben, wenn das View-Fenster gelöscht wird:

```
void
ViewWnd_OnDestroy(HWND hwnd)
{
  . . .                                  // hier weitere Aufräumarbeiten
                                         // für das Dokument
  if (OcRemView)
    OcRemView->ReleaseObjekt();    // ein TOcRemView-Objekt nie löschen

  if (OcDoc) {
    OcDoc->Close();                // Server für eingebettete Teile
                                   // freigeben
    delete OcDoc;                  // dies ist kein COM-Objekt, kann daher
                                   // gelöscht werden
  }
```

```
  if (IsWindow(HwndMain))
    PostMessage(HwndMain, WM_CLOSE, 0, 0);
  HwndView = 0;
}
```

Wenn das View-Fenster gelöscht wird, macht es drei Aufrufe zum Freigeben der Helper-Objekte. *OcRemView->ReleaseObjekt* signalisiert, daß das View-Fenster das Verbinderobjekt *TOcRemView* nicht mehr benötigt. Sie sollten für ein View-Objekt nicht **delete** aufrufen, weil das OLE-System möglicherweise noch weitere Informationen benötigt, bevor es das Löschen des View-Objekts erlaubt. *ReleaseObjekt* erniedrigt einen internen Referenzzähler und trennt das View-Objekt von seinem Fenster. Wenn alle Clients des View-Objekts dieses freigegeben haben, erreicht der Zähler den Stand 0, und das Objekt löscht sich selbst.

Im Gegensatz dazu ist das View-Objekt der Klasse *TOcDocument* kein verbindendes Objekt, Sie können es daher per **löschen** in der normalen Weise löschen. Zuerst sollten Sie jedoch *Close* aufrufen, um die Server-Anwendungen freizugeben, die das OLE-System möglicherweise aufgerufen hat, um in das Dokument des Servers eingebettete Objekte zu unterstützen.

Weil CPPOCF2 niemals mehr als ein Dokument zu einem Zeitpunkt öffnet, werden dort *OcDoc* und *OcRemView* als statische, globale Zeiger deklariert.

```
TOcDocument*  OcDoc       = 0;
TOcRemView*   OcRemView   = 0;
```

Ein Server, der mehrere Clients zugleich mit einer einzigen Instanz der Anwendung unterstützt, oder ein Server, der das Multi-Dokument-Interface (MDI) benutzt, muß für jedes Dokumentfenster ein eigenes Paar von *TOcDocument* und *TOcRemView* erzeugen.

Hinweis

Wenn ein Server gestartet wird, um ein Objekt in einem Container zu unterstützen, so erzeugt er *TOcRemView* an Stelle von *TOcView*, weil er seine Anzeige im einem Remote-Fenster durchführt. Zur Vereinfachung erzeugt CPPOCF2 stets eine Remote View, selbst dann, wenn der Server vom Benutzer direkt gestartet wird. Der einzige damit verbundene Nachteil ist der zusätzliche Aufwand.

Behandlung von WM_OCEVENT

Da die Methode *TOcRemView::SetupWindow* den Verbinder *OcRemView* an das View-Fenster gebunden hat, sendet der Verbinder seine Ereignisbotschaften an das Fenster. Alle ObjectComponents-Ereignisse werden mit der Botschaft WM_OCEVENT gesendet, so daß die Behandlungsroutine des View-Fensters auf WM_OCEVENT antworten muß.

```
long CALLBACK _export
ViewWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
```

```
  switch (message) {
    . . .  // hier andere Routinen zur Extrahierung von Botschaften
    HANDLE_MSG(hwnd, WM_OCEVENT,  ViewWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}
```

Das Makro HANDLE_MSG zum Extrahieren von Botschaften für WM_OCEVENT ist in der Include-Datei ocf/ocfevx.h definiert. In derselben Include-Datei wird eine weitere Extrahierungsroutine zur Verwendung in der Behandlungsroutine der Botschaft WM_OCEVENT definiert.

```
// Unterzuteilungsfunktion für die Botschaft OC_VIEWxxxx
long
ViewWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    // hier  für jede OC_VIEWxxxx-Botschaft, die Sie behandeln wollen,
    // eine Extrahierungsroutine einsetzen
    HANDLE_OCF(hwnd, OC_VIEWCLOSE, ViewWnd_OnOcViewClose);
  }
  return true;
}
```

Die Botschaft WM_OCEVENT führt in *wParam* eine Ereignis-ID mit, genauso, wie die Botschaften WM_COMMAND Befehls-IDs mitführen. OC_VIEWCLOSE ist ein mögliches Ereignis, das anzeigt, das dieser View jetzt geschlossen werden muß. In Anwendungen, die nur ein View-Fenster pro Dokument anzeigen, signalisiert OC_VIEWCLOSE außerdem das Schließen des Dokuments. Das Makro HANDLE_OCF ruft die Behandlungsroutine auf, die Sie jedem ObjectComponents-Ereignis zugeordnet haben, in gleicher Weise, wie HANDLE_MSG die Behandlungsroutine für eine Fensterbotschaft aufruft.

CPPOCF2 behandelt nur die Botschaft OC_VIEWCLOSE. Um andere Botschaften zu behandeln, müssen Sie für jedes Ereignis einen Aufruf des Makros HANDLE_OCF hinzufügen. Eine Liste aller ObjectComponents-Botschaften finden Sie in den Tabellen 19.5 und 19.6.

Behandlung bestimmter View-Ereignisse

Jedes Makro vom Typ HANDLE_OCF ruft eine andere Behandlungsfunktion auf. Im folgenden Beispiel heißt die Behandlungsfunktion *ViewWnd_OnOcViewClose*.

```
bool
ViewWnd_OnOcViewClose(HWND hwnd)
{
  DestroyWindow(hwnd);
  return true;
}
```

Ein Server empfängt diese Botschaft, wenn ein Container das Dokument schließt, das das Objekt des Servers enthält. CPPOCF2 reagiert durch Schließen des View-Fensters. Die Behandlungsroutine für WM_DESTROY erledigt auch das Löschen oder Freigeben der zu dem Server-Dokument gehörenden Helper-Objekte.

Grafische Anzeige des Dokuments

In der Funktion des Servers, die zur grafischen Darstellung dient, ist kein spezieller Code erforderlich. Die Funktion erledigt die Anzeige immer in derselben Weise, unabhängig davon, ob es sich um ein eingebettetes Objekt handelt oder nicht.

```
void
ViewWnd_OnPaint(HWND hwnd)
{
  PAINTSTRUCT ps;
  HDC dc = BeginPaint(hwnd, &ps);
  wsprintf(Buffer, "%u", Counter);
  TextOut(dc, 0, 0, Buffer, lstrlen(Buffer));
  EndPaint(hwnd, &ps);
}
```

Wenn das View-Fenster erzeugt wird, wird ein Zeitgeber gestartet. Jedesmal, wenn das View-Fenster die Botschaft WM_TIMER empfängt, erhöht es den Wert der globalen Variablen *Counter* und ruft *InvalidateRect* auf, damit das Fenster seinen Inhalt selbst neu anzeigt. Bei jedem Aufruf gibt die Paint-Funktion den Wert von *Counter* aus.

4. Programmieren des Hauptfensters

Das View-Fenster führt Aufgaben aus, die sich auf ein einzelnes Dokument beziehen. Das Fenster öffnet und schließt das Dokument und zeigt es auf dem Bildschirm an. Das Rahmenfenster führt Aufgaben für die gesamte Anwendung aus. Es reagiert aus Menübefehle, und es erzeugt und löscht das View-Fenster.

Erzeugen des Hauptfensters

Wenn eine Anwendung ihr Hauptfenster erzeugt, muß sie das Fenster mit ihrem Objekt vom Typ *TOcApp* verbinden. (Das Objekt *TOcApp* ist in der Factory-Callback-Funktion erzeugt worden.)

```
bool
MainWnd_OnCreate(HWND hwnd, CREATESTRUCT FAR* /*lpCreateStruct*/)
{
  if (OcApp)
    OcApp->SetupWindow(hwnd);

  HwndMain = hwnd;
  return true;
}
```

Das Objekt *TOcApp* sendet Botschaften über OLE-Ereignisse an das Hauptfenster der Anwendung. *SetupWindow* teilt *TOcApp* mit, wohin seine Ereignisbotschaften gesendet werden müssen.

Behandlung von WM_OCEVENT

Genau wie das View-Fenster empfängt auch das Rahmenfenster die Botschaft WM_OCEVENT. Das Rahmenfenster empfängt solche Ereignisbotschaften, die die Anwendung als Ganzes betreffen und nicht nur ein bestimmtes Dokument. Die Funktion des Rahmenfensters sendet die Botschaften WM_OCEVENT an eine Zuteilungsroutine, die das jeweilige Ereignis identifiziert und die passende spezielle Behandlungsroutine aufruft. Beide Routinen ähneln sehr dem entsprechenden Code für das View-Fenster.

```
// Standardmäßige Ereignisbehandlungsroutine für das Hauptfenster
long CALLBACK _export
MainWndProc(HWND hwnd, uint message, WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    . . .  // hier andere Routinen zur Extrahierung von Botschaften
    HANDLE_MSG(hwnd, WM_OCEVENT,  MainWnd_OnOcEvent);
  }
  return DefWindowProc(hwnd, message, wParam, lParam);
}

// Zuteilungsunterfunktion für OC_...-Botschaften
long
MainWnd_OnOcEvent(HWND hwnd, WPARAM wParam, LPARAM /*lParam*/)
{
  switch (wParam) {
    HANDLE_OCF(hwnd, OC_APPSHUTDOWN, MainWnd_OnOcViewTitle);
  }
  return true;
}
```

Behandlung einzelner Anwendungsereignisse

Das einzige ObjectComponents-Ereignis, das CPPOCF2 in seinem Hauptfenster behandeln kann, ist OC_APPSHUTDOWN. Ein Server empfängt diese Botschaft, wenn das zuletzt verknüpfte oder eingebettete Objekt geschlossen wird. Wenn der Server vom OLE-System gestartet wurde, kann er sich dann selbst beenden. Wenn der Benutzer den Server direkt gestartet hat, braucht der Server gar nichts zu tun.

```
const char*
MainWnd_OnOcAppShutDown(HWND hwnd)
{
  if (OcRegistrar->IsOptionSet(amEmbedding))
    DestroyWindow(hwnd);
}
```

Das Registrar-Objekt setzt beim Start das Flag *amEmbedding*, falls es den Schalter *-Embedding* in der Kommandozeile der Anwendung findet. Das OLE-System übergibt den Schalter *-Embedding*, wenn es einen Server zur Unterstützung eines verknüpften oder eingebetteten Objekts startet.

5. Erzeugen des Servers

Um den Server zu erzeugen, müssen Sie die passenden Include-Dateien einbinden, ein unterstütztes Speichermodell verwenden und die richtigen Bibliotheken hinzulinken.

Einbinden der Include-Dateien von ObjectComponents

Die folgende Liste zeigt die Include-Dateien, die für einen ObjectComponents-Server erforderlich sind, der ObjectWindows nicht benutzt.

```
#include /ocapp.h       // TOcRegistrar, TOcModule, TOcApp
(Anwendungsverbinder)
#include /ocreg.h       // Registrierungskonstanten und Flags für den
                        // Ablaufmodus der Anwendung
#include /ocdoc.h       // TOcDocument (Verbunddokumentmanager)
#include /ocview.h      // TOcView (Dokument-View-Verbinder)
#include /ocpart.h      // TOcPart (Objekt-Verbinder)
#include /ocremvie.h    // TOcRemView (Dokument-Remote View-Verbinder)
#include /ocfevx.h      // Botschafts-Extrahierer für WM_OCEVENT
```

Compilieren und Linken

ObjectComponents-Anwendungen, die ObjectWindows *nicht* benutzen, können in den Speichermodellen Medium oder Large geschrieben werden. Sie müssen mit den OLE- und den ObjectComponents-Bibliotheken gelinkt werden.

Um CPPOCF0, CPPOCF1 und CPPOCF2 zu erzeugen, wechseln Sie in das Verzeichnis des Programms und geben Sie an der Eingabeaufforderung der Kommandozeile folgendes ein:

```
make MODEL=l
```

Dieses Kommando erzeugt alle drei Programme unter Benutzung des Speichermodells Large.

Die Make-Datei, mit der dieses Beispielprogramm erzeugt wird, greift auf die Datei OCFMAKE.GEN zu.

Funktionsweise der Registrierung

Dieser Abschnitt beschreibt die Funktion der Registriermakros.

Die Makros BEGIN_REGISTRATION und END_REGISTRATION deklarieren eine Struktur, die die Registrierschlüssel und die diesen zugewiesenen Werte speichert. Die Makros, die dazwischen aufgerufen werden,

sind REGDATA und REGFORMAT. Sie fügen jedesmal einen Eintrag in die Struktur ein. Die Hauptstruktur ist vom Typ *TRegList*, jeder einzelne Eintrag in der Struktur vom Typ *TRegItem.* Jeder Eintrag enthält einen Schlüssel und einen Wert für diesen Schlüssel.

```
struct TRegItem {
  char* Key;            // ein standardmäßiger Registrierschlüssel
  TLocaleString Value;  // von Ihnen dem Schlüssel zugewiesener Wert
};
```

Die beiden an REGDATA übergebenen Werte sind ein Schlüssel und ein Wert. Die Makros machen es sehr einfach, der Struktur Schlüssel und Werte hinzuzufügen, ohne selbst die Objekte der Typen *TRegList* und *TRegItem* direkt zu manipulieren. Zur Laufzeit geht ObjectComponents die Tabellen durch und stellt sicher, daß die Informationen korrekt in der Registrierungsdatenbank des Systems gespeichert sind.

Speichern von Informationen in der Registrierungsdatenbank

ObjectComponents kopiert die Informationen aus den Registriertabellen des Programms in die Registrierungsdatenbank des Systems. Das Objekt vom Typ *TOcRegistrar* übernimmt diese Aufgabe bei seiner Erzeugung. Jedesmal, wenn ein Server ein Objekt *TOcRegistrar* erzeugt, überprüft ObjectComponents, ob die Registrierinformation korrekt in der Datenbank gespeichert ist. ObjectComponents vergleicht die aktuellen Werte des Programms für *progid*, *clsid* und dem Programmpfad mit den früher in die Registrierungsdatenbank geschriebenen Werten. Bei irgendeiner Nichtübereinstimmung registriert ObjectComponents das gesamte Programm automatisch neu.

Windows 3.1 speichert die Registrierungsdatenbank in der Datei REG.DAT. Windows NT übergibt sie an die Systemregistrierung, eine Einrichtung, die ausschließlich vom Betriebssystem selbst verwaltet wird. In beiden Fällen verwendet ObjectComponents zum Speichern der Informationen über einen OLE-Server das standardisierte Verfahren. Aus den unter »Erzeugen von Registriertabellen« gezeigten Registriertabellen werden folgende Einträge erzeugt:

```
\CLSID
  {84CA8D41-748F-101B-B3FF-04021C007002} = OWL Drawing Pad
  Insertable =
  Aux UserType =
    2 = Drawing Objekt
  DataFormats =
    GetSet =
      1 = 3,1,32,1
      0 = Embed Source,1,8,1
  verb =
    1 = &Open,0,2
```

```
    0 = &Edit,0,2
  InprocHandler = ole2.dll
  ProgID = DrawingPad
  LocalServer = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE %1
\DrawingPad = OWL Drawing Pad
  Insertable =
  shell =
    open =
      command = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE %1
    print =
      command = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE %1
  protocol =
    StdFileEditing =
      Server = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE
      verb =
        1 = &Open
        0 = &Edit
  CLSID = {84CA8D41-748F-101B-B3FF-04021C007002}

\CLSID
  {84CA8D40-748F-101B-B3FF-04021C007002} = OWL-Zeichenbrettanwendung
  InprocHandler = ole2.dll
  ProgID = DrawingPad.Anwendung.1
  LocalServer = C:\BC45\EXAMPLES\OWL\TUTORIAL\STEP15.EXE
\DrawingPad.Anwendung.1 = OWL-Zeichenbrettanwendung
  CLSID = {84CA8D40-748F-101B-B3FF-04021C007002}
```

Hinweis

Bei der 16-Bit-Version von Windows hat die Registrierungsdatenbank eine maximale Größe von 64 KB. Wenn die Datenbankdatei diese Größe überschreitet, verhält sich das OLE-System unvorhersagbar und es kann erforderlich werden, daß Sie die gesamte Datei REG.DAT löschen müssen. Danach müssen Sie dann alle OLE-Anwendungen in Ihrem System neu registrieren. Sie können jeden ObjectComponents-Server registrieren oder seine Registrierung löschen, indem Sie den Schalter *-RegServer* bzw. *-UnregServer* in seine Kommandozeile schreiben. Das Löschen der Registrierung unbenutzter Anwendungen oder veralteter Versionen ist eine gute Möglichkeit, in der Datenbank Platz zu sparen.

Um mehr über die Registrierungsdatenbank zu erfahren, sehen Sie unter dem Thema »Registrierungsdatenbank« in der Online-Hilfe zur Programmierung unter der 16-Bit-Version von Windows nach (WIN31WH.HLP).

Registrieren lokalisierter Einträge

Die den Registrierparametern zugewiesenen Werte müssen oftmals lokalisiert werden. Der Abschnitt »Lokalisieren von Symbolnamen« erklärt die Erzeugung von Ressourcen LOCALIZATION, die Übersetzungstabellen für Ihre OLE-Strings enthalten. Außer dem Einfügen von Übersetzungen in Ihre Ressourcen müssen Sie die Strings auch markieren, damit ObjectComponents weiß, welche davon als lokalisierte Versionen existieren.

```
REGDATA(description, "@myapp_description")
```

Das Präfix @ teilt ObjectComponents mit, daß der String *myapp_description* ein Bezeichner für eine Ressource LOCALIZATION steht, wobei die echte Beschreibung in mehreren Sprachen gespeichert ist. Für weitere Informationen über das Lokalisieren von OLE-Strings siehe den Eintrag *TLocaleString* im *ObjectWindows Referenzhandbuch*.

Registrieren von anwendungsspezifischen Einträgen

Das Makro REGDATA verknüpft Strings mit Schlüsseln. Das Registrar-Objekt geht die Schlüsselliste durch und schreibt, soweit erforderlich, die zugehörigen Strings in die Registrierungsdatenbank des Systems. Die standardmäßigen Schlüssel wie etwa *progid* und *clsid* korrespondieren mit Standardeinträgen in der Datenbank. Wenn Sie Werte für Schlüssel registrieren wollen, die nicht dem Standard entsprechen, müssen Sie dazu das Makro REGITEM benutzen. Der erste Parameter für REGITEM ist der vollständige Schlüssel in genau der Form, wie Sie ihn an eine Funktion wie etwa *RegOpenKey* übergeben würden.

```
REGITEM("CLSID\\<clsid>\\Conversion\\Readable\\Main","FormatX,FormatY")
```

Wie das Beispiel zeigt, können die Argumente für REGITEM auch eingebettete Parameternamen, eingeschlossen in spitze Klammern, enthalten. Wenn ObjectComponents diesen Eintrag registriert, so ersetzt es zuerst den Ausdruck <*clsid*> mit dem (beliebigen) Wert, der in dem Registrieranweisungsblock dem Parameter *clsid* zugewiesen wurde.

Für Informationen über *RegOpenKey* siehe die Hilfedatei für das Windows API.

Erzeugen eines DLL-Servers

Im Normalfall ist ein OLE-Server ein unabhängiges EXE-Programm, das vom Benutzer direkt gestartet oder indirekt von einem OLE-Container aufgerufen werden kann. Sie können aber einen OLE-Server auch als DLL implementieren. Ein als DLL programmierter Server wird manchmal auch »In Prozeß-Server« genannt, weil der DLL-Code im selben Prozeß abläuft wie der des Clients. Die Ausdrücke »EXE-Server« und »Server-Anwendung« beziehen sich speziell auf einen Server, der als EXE-Datei realisiert ist. ObjectComponents erlaubt es Ihnen, sowohl EXE- als auch DLL-Server zu erzeugen. Wenn Sie ObjectWindows benutzen, erfordert die Umwandlung zwischen diesen beiden Arten nur zwei einfache Änderungen.

Hinweis

Die folgenden Betrachtungen und Anleitungen gelten sowohl für Automations-Server als auch für OLE-Server.

Vor- und Nachteile von DLL-Servern

Der Hauptvorteil eines DLL-Servers ist seine Ablaufgeschwindigkeit. Da ein DLL-Server im Adreßraum des Containers abläuft, wird er schnell geladen und reagiert schnell. Ein EXE-Server dagegen ist ein eigener Prozeß und erfordert daher bei der Zusammenarbeit mit einem Container eine gewisse Kommunikation zwischen den Prozessen. Das OLE-System bringt die Prozeß-zu-Prozeß-Aufrufe in eine Warteschlange und ordnet die Funktionsaufrufe mit ihren Parametern, indem es sie in das geeignete Format für das Prozeßkommunikationsprotokoll bringt. (Das dabei benutzte Protokoll heißt LRPC, »Lightweight Remote Procedure Call«.) Der Vorgang der Bildung einer Warteschlange von Zeichenbefehlen in einer Metadatei ist besonders langsam, so daß ein DLL-Server beim Erzeugen grafischer Darstellungen von OLE-Objekten erhebliche Zeitvorteile bietet.

Die Benutzung eines DLL-Servers hat aber auch einige Nachteile. Während das OLE-System die Zusammenarbeit zwischen 16-Bit- und 32-Bit-EXE-Anwendungen unterstützt, kann eine 16-Bit-Windows Anwendung keinen 32-Bit-DLL-Server und eine Win32-Anwendung keinen 16-Bit-DLL-Server benutzen. Außerdem besitzen DLLs keine Botschaftswarteschlangen. Als Folge davon kann ein DLL-Server nicht ohne weiteres eine Aufgabe im Hintergrund ausführen. ObjectWindows überwindet diese Einschränkung durch Verwendung eines Zeitgebers, so daß es trotzdem noch die *IdleAction*-Methoden von Objekten, die *TAnwendung* oder *TOleFrame* abgeleitet sind, aufrufen kann. (ObjectWindows benutzt den Zeitgeber außerdem für interne Aufgaben wie die Befehlsfreigabe für von Mauspaletten, für das Löschen von nicht mehr aktuellen Fenstern und für die Programmfortsetzung nach Eintreten von Exceptions.)

Da ein DLL-Server Teil des Container-Prozesses wird, können sich Fehler in einem der Programme auf das andere auswirken, so daß das Debuggen bei einem DLL-Server manchmal schwieriger ist.

DLL-Server haben auch das Problem der Benutzerschnittstelle. Wenn zum Beispiel ein Container eine Bearbeitungssitzung mit einem Server startet, so spielt es für den Benutzer keine Rolle, ob es sich um einen EXE- oder einen DLL-Server handelt; die Benutzerschnittstelle für das Starten dieser Sitzung ist in beiden Fällen identisch. Die Lebensdauer eines DLL-Servers ist aber abhängig vom Container, der den Server geladen hat. Beim Programmende des Containers wird die Server-DLL aus dem Speicher entfernt. Dies kann zu Problemen führen, wenn die Benutzerschnittstelle des Servers es dem Benutzer, wie es normal ist, erlaubt, mehrere Dokumente gleichzeitig zu bearbeiten. Wenn der Benutzer ein neues Dokument erzeugt, während er ein eingebettetes Objekt bearbeitet, so könnte er die Bearbeitung des neuen Dokuments auch dann fortsetzen wollen, nachdem das Container-Programm geendet hat; der Server ist dann aber gar nicht mehr im Speicher. Dies ist ein spezielles Problem für MDI-Serv-

er, da die MDI-Schnittstelle es den Benutzern erlaubt, in einer einzigen Sitzung mehrere Dokumente zu öffnen. Typische DLL-Server erlauben daher nicht die gleichzeitige Bearbeitung mehrerer Dokumente.

Schließlich haben DLL-Server noch einen weiteren Nachteil. Das OLE 2 System stellt zwar eine Kompatibilitätsschicht bereit, über die OLE 2 Server mit OLE 1 Clienten zusammenarbeiten können, aber diese Schicht arbeitet nur mit EXE-Servern. Ein DLL-Server kann daher keinen OLE 1 Client unterstützen.

Erzeugen eines DLL-Servers

Die Umwandlung eines ObjectWindows EXE-Servers in einen DLL-Server erfordert nur wenige Modifikationen. Die folgende Liste beschreibt sie in Kurzfassung. Die darauffolgenden Abschnitte enthalten weitere Detailinformationen.

1. Aktualisieren Sie Ihre Dokumentregistrierstruktur.
2. Erzeugen Sie das Programm bei aktivierter DLL-Option. Linken Sie die DLL mit kompatiblen Bibliotheken von ObjectComponents und ObjectWindows hinzu.

Aktualisieren Ihrer Dokumentregistrierstruktur

Die Dokumentregistrierstrukturen des DLL-Servers müssen den Schlüssel *serverctx* mit dem Stringwert *Inproc* enthalten. Dadurch wird es ObjectComponents erlaubt, Ihre Anwendung als DLL-Server mit OLE-Fähigkeiten zu registrieren. EXE-Server erfordern die Benutzung des Schlüssels *serverctx* nicht, da ObjectComponents einen Server standardmäßig als EXE-Server registriert.

Der folgende Code illustriert die Dokumentregistrierstruktur eines DLL-Servers. Er stammt aus dem Beispielprogramm »Tic Tac Toe« in EXAMPLES/OWL/OCF/TTT.

```
BEGIN_REGISTRATION(DocReg)
 REGDATA(progid,      "TicTacToeDll")
 REGDATA(description,"TicTacToe DLL")
 REGDATA(serverctx, "Inproc")
 REGDATA(menuname,    "TicTacToe-Spiel")
 REGDATA(insertable, "")
 REGDATA(extension,  "TTT")
 REGDATA(docfilter,  "*.ttt")
 REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
 REGDATA(verb0,       "&Play")
 REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
 REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

Um Ihren EXE-Server in einen DLL-Server umzuwandeln, müssen Sie Ihre Anwendungsregistrierstruktur nicht unbedingt modifizieren. Es ist

aber empfehlenswert, unterschiedliche Werte für *clsid* und *progid* zu benutzen, besonders dann, wenn Sie beabsichtigen, häufig zwischen den beiden Arten zu wechseln. Sie können das Vorhandensein des Makros BI_APP_DLL prüfen, um eine Registrierstruktur zu deklarieren, die sowohl für DLL- als auch für EXE- Server geeignet ist. Das Makro ist nur definiert, wenn Sie gerade eine DLL erzeugen. Der folgenden Code zeigt ein Beispiel für eine Dokumentregistrierung, wobei zwei Sätze von Werten für *progid* und *clsid* benutzt werden.

```
REGISTRATION_FORMAT_BUFFER(100)

// Anwendungsregistrierstruktur
BEGIN_REGISTRATION(AppReg)
#if defined(BI_APP_DLL)
 REGDATA(clsid,       "{029442B1-8BB5-101B-B3FF-04021C009402}")// DLL clsid
 REGDATA(progid,      "TicTacToe.DllServer")                   // DLL progid

#else
 REGDATA(clsid,
"{029442C1-8BB5-101B-B3FF-04021C009402}") // EXE clsid
 REGDATA(progid,      "TicTacToe.Anwendung")                   // EXE progid
#endif
 REGDATA(description,"TicTacToe-Anwendung")               // Beschreibung
END_REGISTRATION

// Dokument registrier structure
BEGIN_REGISTRATION(DocReg)
#if defined(BI_APP_DLL)
 REGDATA(progid,      "TicTacToeDll")
 REGDATA(description,"TicTacToe DLL")
 REGDATA(serverctx, "Inproc")
#else
 REGDATA(progid,      "TicTacToe.Game.1")
 REGDATA(description,"TicTacToe-Spiel")
 REGDATA(debugger,    "tdw")
 REGDATA(debugprogid,"TicTacToe.Game.1.D")
 REGDATA(debugdesc,   "TicTacToe-Spiel (Debug-Version)")
#endif
 REGDATA(menuname,    "TicTacToe-Spiel")
 REGDATA(insertable,  "")
 REGDATA(extension,   "TTT")
 REGDATA(docfilter,   "*.ttt")
 REGDOCFLAGS(dtAutoDelete | dtUpdateDir | dtCreatePrompt | dtRegisterExt)
 REGDATA(verb0,       "&Play")
 REGFORMAT(0, ocrEmbedSource,  ocrContent, ocrIStorage, ocrGet)
 REGFORMAT(1, ocrMetafilePict, ocrContent, ocrMfPict,   ocrGet)
END_REGISTRATION
```

Beachten Sie, daß die Debug-Schlüssel (*debugger, debugprogid* und *debugdesc*) nicht benutzt werden, wenn ein DLL-Server erzeugt wird. Sie sind nur wirksam, wenn Sie einen EXE-Server erzeugen, den ein Debugger laden kann. (Weitere Information zum Debuggen eines DLL-Servers siehe unter »Debuggen eines DLL-Servers«).

Compilieren und Linken

ObjectWindows DLL-Server müssen im Speichermodell Large compiliert werden. Sie müssen mit den OLE-, ObjectComponents- und ObjectWindows-Bibliotheken gelinkt werden.

Die integrierte Entwicklungsumgebung (IDE) wählt automatisch die passenden Optionen für die Erzeugung aus, wenn Sie als Ziel *Dynamic Library* angeben und in der Liste der Standardbibliotheken die OWL- und OCF-Unterstützung wählen. Sie können wahlweise die statische oder dynamischen Version der Standardbibliotheken wählen.

Wollen Sie einen ObjectWindows-DLL-Server von der Kommandozeile aus erstellen, so legen Sie sich eine kurze Make-Datei an, die die Include-Datei OWLOCFMK.GEN im Unterverzeichnis EXAMPLES einbindet. Das folgende Beispiel zeigt eine Make-Datei zum Erstellen des Beispielprogramms »Tic Tac Toe«:

```
MODELS=ld
SYSTEM = WIN16

DLLRES = ttt
OBJDLL = ttt.obj

!include $(BCEXAMPLEDIR)\owlocfmk.gen
```

DLLRES speichert den Basisnamen Ihrer Ressourcendatei und den Namen der fertigen DLL. OBJDLL speichert die Namen der Objektdateien, aus denen das Beispielprogramm zusammengestellt werden soll. Schließlich sollte Ihre Make-Datei auch die Include-Datei OWLOCFMK.GEN einbinden.

Nennen Sie Ihre Datei MAKEFILE, und geben Sie folgendes an der Eingabeaufforderung der Kommandozeile ein:

```
make MODEL=l
```

Das Hilfsprogramm MAKE stützt sich auf die Befehle in OWLOCFMK.GEN und erzeugt eine neue Make-Datei, die auf Ihr Projekt zugeschnitten ist. Der Befehl bewirkt auch das Abarbeiten der neuen Make-Datei und somit die Erzeugung Ihres Servers. Wenn Sie den Befehl auf `MODEL=d` ändern, so erzeugt der obige Befehl eine neue Make-Datei und erstellt Ihre DLL unter Benutzung der DLL-Versionen der Bibliotheken an Stelle der statischen Bibliotheken für das Speichermodell Large.

Weitere Informationen über die Verwendung von OWLOCFMK.GEN finden Sie bei den Anweisungen am Anfang der Datei MAKEFILE.GEN im Unterverzeichnis EXAMPLES.

Debuggen eines DLL-Servers

Die für das Debuggen von DLLs benutzten allgemeinen Techniken sind ebenso auf DLL-Server anwendbar. Die folgenden Schritte beschreiben eine Vorgehensweise bei der der Turbo Debugger for Windows zum Setzen von Haltepunkten in einem DLL-Server benutzt wird.

1. Erzeugen und registrieren Sie den DLL-Server.

   - Erzeugen Sie den Server mit eingefügten Debug-Informationen.
   - Registrieren Sie den Server unter Benutzung des Hilfsprogramms REGISTER.EXE.
   - Vergewissern Sie sich, daß die Registrierung erfolgreich verlief, indem Sie das Programm REGEDIT starten und sich die Dateitypen für Ihren Server ansehen. (REGEDIT ist der bei Windows mitgelieferte Registriereditor.)

2. Starten Sie Turbo Debugger for Windows und laden Sie einen Container.

   - Wählen Sie den Menüpunkt *File/Open*, geben Sie den Namen des Containers in das Feld für den Programmnamen ein, und klicken Sie auf OK.

Der Debugger lädt nun den Container. Wenn der Container ohne eingefügte Debug-Informationen erzeugt worden ist, so erhalten Sie möglicherweise eine Warnung, die Sie aber ohne Gefahr ignorieren können.

Laden Sie die Debug-Informationen des Server.

- Wählen Sie *View/Module* im Hauptmenü des Debuggers. Dadurch wird das Dialogfenster »*Load module source or DLL symbols*« (Sie können den Moduldialog auch durch Betätigen der Taste [*F3*] aufrufen).
- Geben Sie den kompletten Namen der DLL-Server-Datei in das Feld *DLL Name* ein.
- Wählen Sie die Option *Yes* im Feld für die Startoptionen des Debuggers, und klicken Sie auf *Add DLL*. Der Name Ihres DLL-Servers (gefolgt von *!!*) erscheint nun als ausgewählter Eintrag in der Liste *DLLs & Programs*.
- Klicken Sie auf *Symbol Load*.

Wenn Sie eine die Fehlermeldung erhalten, daß die DLL nicht geladen wurde, so betätigen Sie die Taste [*Esc*], um zum Hauptmenü des Debuggers zurückzukehren, und gehen Sie dann zum folgenden Schritt. Andernfalls überspringen Sie den folgenden Schritt und fahren mit Schritt 5 fort.

Hinweis

Wenn Sie keine die Fehlermeldung, daß die DLL nicht geladen wurde, erhalten haben, so befand sich Ihr DLL-Server bereits im Speicher, bevor der Container ihn aktivierte. Dies kann vorkommen, wenn ein anderer Container gerade mit einem der Objekte Ihres Servers arbeitet. In den meisten Fällen bedeutet es aber, daß Ihr Server in einer früheren Sitzung abgestürzt ist oder nicht korrekt beendet wurde.

Starten Sie den Container und fügen sie eines der Objekte Ihres Servers ein.

- Wählen Sie den Menüpunkt *Run/Run* (oder betätigen Sie die Taste [*F9*], um den Container zu starten.
- Wählen Sie im Container den Menüpunkt *Bearbeiten/Einfügen/Objekt* und fügen Sie ein Objekt Ihres Servers ein.

Der Debugger erscheint wieder, sobald das OLE-System Ihren DLL-Server geladen hat.

Lassen Sie die Quelltextmodule der DLL anzeigen, und setzen Sie Haltepunkte.

- Wählen Sie im Hauptmenü des Debuggers den Menüpunkt *View/Module*, um die Namen der Quelltextmodule zu sehen, aus denen Ihr Server erzeugt wurde. Die Dateinamen erscheinen in der Liste *Source Modules*.
- Wählen Sie Quelltextmodule durch Doppelklicken auf den Dateinamen aus.
- Setzen Sie Haltepunkte in Ihrem Server.
- Wählen Sie den Menüpunkt *Run/Run* (oder betätigen Sie die Taste [*F9*]), um die Steuerung an die Container-Anwendung zurückzugeben.

Falls Sie Schritt 4 übersprungen haben, fügen Sie jetzt eines der Objekte Ihres Servers in das Container-Dokument ein.

Der Debugger wird den Ablauf an den in Ihren Quelltextdateien gesetzten Haltepunkten stoppen und es Ihnen ermöglichen, schrittweise durch Ihr Server-Programm zu gehen, Werte von Variablen einzusehen und die Logik Ihres Codes zu überprüfen.

Tools für DLL-Server

Bevor Sie Ihren DLL-Server starten können, müssen Sie seine Registrierinformationen in der Registrierungsdatenbank des Systems eintragen. Das Hilfsprogramm REGISTER erledigt dies für Sie. Ein anderes Hilfsprogramm ist DLLRUN, es ermöglicht Ihnen, Ihren DLL-Server jederzeit als unabhängige Anwendung laufen zu lassen, was zum Testen manchmal praktisch sein kann.

Registrieren Ihres DLL-Servers

Das Hilfsprogramm REGISTER.EXE registriert einen ObjectComponents-DLL-Server. Geben Sie dem Programm dazu in der Kommandozeile den

Namen Ihres Servers gefolgt von dem Schalter *-RegServer* an. Hier folgt das Kommando zum Registrieren von »Tic Tac Toe«:

```
register ttt.dll -RegServer
```

Das Hilfsprogramm REGISTER ist eine Windows Anwendung und keine DOS Anwendung, deshalb müssen Sie es aus Windows heraus aufrufen. Das Hilfsprogramm WINRUN ermöglicht es Ihnen, Windows Anwendungen von der DOS Eingabeaufforderung aus zu starten. (Weitere Informationen über WINRUN siehe die Textdatei UTILS.TXT.)

REGISTER kann auch die Registrierung Ihres Servers löschen. Bei Löschen einer Registrierung werden alle Ihren Server betreffenden Einträge aus der Registrierungsdatenbank gelöscht. Es ist empfehlenswert, die Registrierung einer Programmversion zu löschen, bevor Sie eine die Folgeversion registrieren. Zum Löschen einer Registrierung benutzen Sie den Schalter *-UnregServer*. Das folgende Kommando löscht die Registrierung von »Tic Tac Toe«:

```
register ttt.dll -UnregServer
```

Starten Ihres DLL-Servers

Das Hilfsprogramm DLLRUN.EXE ermöglicht es Ihnen, einen ObjectComponents-DLL-Server so zu starten, als wäre er ein unabhängig ausführbares Programm. Die Möglichkeit der Ausführung in diesem Modus ist zum Debuggen nützlich. Außerdem haben Ihre Kunden dann die Wahlmöglichkeit, Ihren Server in beiden Modi laufen zu lassen, ohne daß Sie dazu zwei Versionen derselben Anwendung vertreiben müssen.

Übergeben Sie DLLRUN in der Kommandozeile den String *progid* des Servers. Dabei handelt es sich um den Wert, der dem Schlüssel *progid* in der Registriertabelle des Servers zugewiesen wurde. Das folgende Kommando startet den Server »Tic Tac Toe«:

```
dllrun TicTacToeDll
```

DLLRUN startet den DLL-Server im Ablaufmodus eines EXE-Programms. Der Ablaufmodus einer ObjectComponents-Anwendung wird durch einen Satz von Einzelbit-Flags repräsentiert, die Sie durch Aufrufen von *TOcModule::IsOptionSet* abfragen können. (Beachten Sie, daß das Anwendungsobjekt eines OLE-Programms von *TAnwendung* und von *TOcModule* abgeleitet ist.)

Die Flags für den Ablaufmodus sind im Aufzählungstyp *TOcAppMode* definiert. *amEmbedded* ist gesetzt, wenn der Server vom OLE-System, nicht vom Benutzer aufgerufen wurde. *amExeModule* ist gesetzt, wenn die Anwendung als EXE-Datei erzeugt wurde. *amExeMode* ist gesetzt, wenn die Anwendung unabhängig (wie eine EXE-Datei) läuft, selbst wenn sie als DLL erzeugt wurde.

Der folgende Code prüft die Flags, um den Ablaufmodus des Servers festzustellen:

```
void
TMyApp::TestMode()
{
  if (IsOptionSet(amExeMode))            // Server läuft als EXE?
    if (!IsOptionsSet(amExeModule)) {    // wenn ja, ist er als EXE
erzeugt?
      // Server ist eine DLL, die im EXE-Modus läuft
    } else {
      // Server wurde als EXE-Datei erzeugt
  } else {
    // Server ist eine DLL, die im Prozeß eines Clients läuft
}
```

K A P I T E L 22

Automatisierung von Anwendungen

Ein Programm zu automatisieren bedeutet, seine Funktionen anderen Programmen zugänglich zu machen. Ein automatisiertes Programm kann über OLE von anderen Programmen gesteuert werden; die betreffenden Befehle werden über OLE geschickt. Die Schnittstelle zur OLE-Automatisierung ist ObjectComponents. Über ObjectComponents kann man beliebige C++ Klassen für OLE zugänglich machen, ohne bestehende Klassen neu strukturieren zu müssen.

Der Automatisierungsvorgang verläuft immer gleich und ist unabhängig davon, ob das Anwendung ObjectWindows verwendet. Die im vorliegenden Kapitel beschriebenen Schritte gelten für jedes C++ Programm.

Das vorliegende Kapitel beschreibt, wie man:

- Anwendungen automatisiert
- Aufzählungswerte zugänglich macht
- einen Hook zur Überprüfung von Befehlsargumenten einrichtet
- mehrere C++ Objekte zu einem einzigen automatisierten OLE-Objekt zusammenfaßt
- ein Automatisierungsobjekt nach seiner Löschung unzugänglich macht
- landessprachlich Strings entsprechend der Spracheinstellung des Controllers verwendet
- eine Kollektion von Objekten zugänglich macht
- eine Typbibliothek erzeugt und registriert

Automatisierungs-Server können als DLLs erzeugt werden, und zwar nach denselben Methoden wie bei der Erzeugung eines programminternen Servers zum Verknüpfen und Einbetten. Siehe “Erzeugung eines DLL-Servers” in diesem Kapitel.

Vorgehensweise zur Automatisierung einer Anwendung

Zur Automatisierung eines Programms sind folgende Schritte erforderlich, die anschließend im einzelnen erläutert werden:

Registrierung der Anwendung

- Erzeugung einer Registriertabelle mittels Registriermakros
- Erzeugung eines Registrierobjektes und Aufruf seiner *Run*-Methode

Deklaration der zugänglich gemachten Methoden und Eigenschaften in einer Deklarationstabelle

- Bei Bedarf Erstellung von Hook-Funktionen zum Aufzeichnen, Filtern und Widerrufen von Befehlen und zur Überprüfung von Befehlsargumenten

Zuweisung von öffentlichen Namen für die bereitzustellenden Methoden und Eigenschaften über eine Definitionstabelle

Erzeugung der Anwendung

- Einbindung von Automatisierungs-Header-Dateien in den Quelltext
- Verknüpfung mit den Bibliotheken von OLE und ObjectComponents

Manchmal können zwei weitere Schritte zweckmäßig sein:

- Bereitstellung einer Ressourcentabelle mit übersetzten Strings, damit OLE sich an die vom Anwender gewählte Sprache anpassen kann.
- Benachrichtigung von OLE, wenn das Objekt gelöscht wird. Dies ist eine reine Sicherheitsmaßnahme und meistens nicht erforderlich.

Über ObjectComponents kann OLE Elemente Ihrer Klassen mittels standardmäßiger C++ Mechanismen erreichen. Zur Laufzeit können andere Programme über OLE Befehle an Ihr Programm senden.

Ein Programm, das sich für den Empfang von Automatisierungsbefehlen bereithält, wird als Automatisierungs-Server bezeichnet; umgekehrt ist ein Programm, das Befehle an andere Programme richtet, ein *Automatisierungs-Kontroller.*

Zahlreiche Code-Beispiele im vorliegenden Kapitel basieren auf dem im Verzeichnis `EXAMPLES/OCF/AUTOCALC` installierten Beispielprogramm *AutoCalc*. *AutoCalc* zeichnet einen Taschenrechner auf den Bildschirm; der Benutzer kann auf diesem Rechner durch Anklicken von Schaltflächen Rechenoperationen ausführen. *AutoCalc* automatisiert außerdem seine Klassen und ist somit für Befehle eines Kontrollerprogramms in der gleichen Weise ansprechbar wie über das direkte Anklicken der Rechnertasten.

Registrierung eines Automatisierungs-Servers

Beim Registrieren einer Anwendung werden OLE Informationen über die Fähigkeiten dieses Programms übermittelt. Zunächst erfaßt man die betreffenden Informationen in einer Registriertabelle und leitet diese dann an den Konstruktor eines Registrarobjektes weiter.

Erzeugung einer Registriertabelle

Ein Automatisierungs-Server muß in der Registriertabelle des Anwendungs vier Informationen bereitstellen: die Programm-ID, die Klassennamen-ID, eine Eigenbeschreibung und Kommandozeilenargumente für den Aufruf des Automatisierungs-Servers.

```
BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,       "{877B6200-7627-101B-B87C-0000C057CE4E}")
  REGDATA(progid,      "Calculator.Application")
  REGDATA(description,"Automated Calculator 1.0 Application")
  REGDATA(cmdline,     "/Automation")
  REGDATA(version,     "1.2")
END_REGISTRATION
```

Die Registriermakros erzeugen in diesem Beispiel eine Struktur namens

AppReg. Der Parameter *clsid* muß eindeutig sein und deshalb eigens erzeugt werden; Hinweise hierzu finden sich im *ObjectWindows Referenzhandbuch* unter dem Abschnitt über den Registrierschlüssel *clsid*. Der *progid* für einen Automatisierungs-Server besteht üblicherweise aus zwei Teilen: dem eigentlichen Programmnamen (hier: "Calculator") und dem Objekttyp (hier: "Application"). Als Trennzeichen in *progid* muß ein Punkt (".") verwendet werden.

Wenn ein Server mehrere Arten von automatisierbaren Objekten erzeugt, muß jedes einen eigenen *progid* erhalten, z.B. *AppName.MySecondObject* und *AppName.MyThirdObject*. Ein *clsid* braucht jedoch nur für die erste Objektart generiert zu werden; *ObjectComponents* inkrementiert dann auf dieser Basis selbständig *clsid* für jedes weitere Objekt.

Der *progid* ist für Benutzer, die Ihr Objekt in ihren Automatisierungsskripts erzeugen, sichtbar, ebenso wie der *description*-String, wenn OLE das System nach verfügbaren Automatisierungsobjekten durchsucht.

Beim letzten *cmdline*, muß der Automatisierungs-Server in der Regel den Schalter *-Automation* angeben. Wenn ein Automatisierungs-Kontroller Ihr Objekt anfordert, ruft OLE Ihre Anwendung auf und plaziert den Wert *cmdline* in die Kommandozeile.

Obwohl die OLE-Konventionen diesen Schalter vorschreiben, wird er von OLE selbst nicht beachtet, wohl aber von ObjectComponents - hiermit legt es fest, ob für die einzelnen OLE-Clients jeweils neue Instanzen Ihres Programms aufzurufen sind oder ob eine einzelne Instanz alle Cli-

ents bedient. Normalerweise sollte man vermeiden, daß mehrere verschiedene Clients Befehle an ein und dasselbe Objekt senden können, weil diese Befehle allzu leicht miteinander kollidieren. Wenn der Schalter *-Automation* angegeben ist, läßt ObjectComponents nur einen Client pro Programminstanz zu. (Der Schalter *-Automation* hat Vorrang vor *ocrMultipleUse*, soweit in Ihrem Schalter *usage* angegeben.) Wenn Ihr Automatisierungs-Server ausdrücklich mehrere Clients gleichzeitig mit einer einzelnen Instanz unterstützen soll, registrieren Sie ihn als *ocrMultipleUse* und unterlassen die Angabe *-Automation* im String *cmdline*.

Der Schalter *-Automation* veranlaßt die Erzeugung einer OLE-Factory; diese wird von ObjectComponents registriert und steht dann zur Laufzeit für OLE-Aufrufe zur Verfügung, um das gewünschte Server-Objekt zu erzeugen. Wenn der Schalter angegeben ist, registriert ObjectComponents die Anwendungdklasse, damit OLE eine Instanz des Programms erzeugen kann. Die Dokument-Factory wird von ObjectComponents nicht registriert. Soll die Anwendung nur einmal benutzt werden, so löscht ObjectComponents die Factory nach der Erzeugung der ersten Programminstanz.

Die Registriertabelle im Beispiel enthält Informationen über das Anwendung und heißt deshalb Anwendungsregistriertabelle. Mit denselben Makros kann ein Programm auch *Dokumentregistrierstrukturen erzeugen, in denen die vom Server erzeugten Dokument- bzw. Objektarten beschrieben werden. Ein Automatisierungs-Server erzeugt automatisierte Dokumente, wenn die Kontroller das automatisierte Objekt vor der Befehlserteilung einbetten sollen. Weitere Hinweise zu Dokument- und Anwendungsregistrierstrukturen finden sich im Abschnitt "Erzeugung von Registriertabellen".*

Tabelle 22.1 gibt eine Kurzübersicht über alle Registrierschlüssel, die ein Automatisierungs-Server benötigen könnte. Sie zeigt, welche Schlüssel wahlfrei und welche obligatorisch sind und ob sie in die Anwendungsregistriertabelle (üblicher Name: AppReg) oder Dokumentregistriertabelle (üblicher Name: *DocReg) gehören.*

Tabelle 22.1: Vom Automatisierungs-Server registrierte Schlüssel

Schlüssel	In AppReg?	In DocReg?	Beschreibung
clsid	Ja	Wahlfrei	Global eindeutiger Bezeichner (GUID); wird automatisch für die Struktur *DocReg* erzeugt
description	Ja	Ja	Maximal 40stellige Beschreibung
progid	Nein	Ja	Eindeutiger Name des Programms bzw. Objekttyps
extension	Nein	Wahlfrei	Erweiterung der dem Server zugeordneten Dokumentdateierweiterung
docfilter	Nein	Ja, sofern nicht *dtHidden*	Filter für Dialogfeld *Datei öffnen* (Jokerzeichen zulässig)
docflags	Nein	Ja	Optionen für die Ausführung des Dialogfeldes Datei öffnen

typehelp	Nein Wahlfrei	Name einer .HLP-Datei für unterstützte Befehle	
helpdir	Nein Wahlfrei	Pfad einer .HLP-Datei für unterstützte Befehle (Vorgabewert: Pfad des aktuellen Moduls)	
debugger	Nein Wahlfrei	Kommandozeile für Debugger-Lauf	
debugprogid	Nein Wahlfrei	(Eindeutiger) Name der Debug-Version	
debugdesc	Nein Wahlfrei	Beschreibung der Debug-Version	
cmdline	Wahlfrei	Nein	Argumente für die Kommandozeile des Servers
path	Wahlfrei	Nein	Pfad der Serverdatei (Vorgabewert: Pfad des aktuellen Moduls)
permid	Nein Wahlfrei	Namen ohne Versionsangabe	
permname	Nein Wahlfrei	Beschreibung ohne Versionsangabe	
usage	Wahlfrei	Wahlfrei	Unterstützung für parallel ausgeführte Clients
language	Wahlfrei	Nein	Sprache für registrierte Strings (Vorgabewert: eingestellte Anwendersprache des Systems)
version	Wahlfrei	Nein	Haupt- und Unterversionsnummer (Vorgabewert: “1.0”)

Die Tabelle setzt voraus, daß die Server-Dokumente Automatisierung unterstützen. Ist dies nicht der Fall, braucht der Server nur *docflags* und *docfilter* zu registrieren.

Ist der Server gleichzeitig ein OLE-Container oder ein Verknüpfungs- und Einbettungs-Server, so sollten Sie auch die Container-Tabelle bzw. Server-Tabelle zu Rate ziehen. Es sind alle Schlüssel zu registrieren, die in den für das Anwendung in Frage kommenden Tabellen benötigt werden.

Weitere Hinweise über die einzelnen Registrierschlüssel und ihre Werte finden Sie unter dem betreffenden Schlüsselnamen im *ObjectWindows Referenzhandbuch*.

Wenn ein Automatisierungs-Server unterschiedliche Systemsprachen unterstützt, sollte er den von ihm registrierten String *description* an die Landessprache anpassen. (Die *progid* darf nie angepaßt werden!) Weitere Hinweise zur Lokalisierung finden sich in diesem Handbuch im Abschnitt “Lokalisierung von Symbolnamen”.

Die komplette Struktur *AppReg* wird anschließend an das Programmobjekt von *TRegistrar* übergeben und in das Register eingetragen.

Erzeugung eines Registrarobjektes

Ein Automatisierungs-Server benötigt ebenso wie Verknüpfungs- und Einbettungsanwendungen ein Registrarobjekt. Anwendungen jedoch, die nur eine Automatisierung (ohne Verknüpfen und Einbetten) unterstützen, sollten *TRegistrar* statt *TOcRegistrar* erzeugen. *TRegistrar* ist die Basisklasse für *TOcRegistrar*. *TOcRegistrar* erweitert *TRegistrar* - es verbindet die Anwendung mit den Schnittstellen zur Unterstützungsbibliothek BOCOLE , die Verknüpfen und Einbetten unterstützen.

Deklarieren Sie zunächst einen statischen Zeiger zur Aufnahme von *TRegistrar**. Sorgen Sie mit dem Template > (Template)*TPointer*<> dafür, daß das Registrarobjekt am Programmende ordnungsgemäß gelöscht wird.

```
TPointer<TRegistrar> Registrar;  // in WinMain or LibMain initialisiert
```

Erstellen Sie in der Hauptprozedur (für *AutoCalc* ist dies *WinMain*) das Registrarobjekt, und rufen Sie seine Methode *Run* auf.

```
try {

::Registrar=new TRegistrar(AppReg,TOcAutoFactory<TCalc>,string(cmdLine),hIn
st)
  if (!::Registrar->IsOptionSet(amAnyRegOption))
    ::Registrar->Run();
    ::Registrar = 0; // löscht Registrar durch Ersetzen des Zeigers
  return 0;
}
catch (TXBase& x) {
  ::MessageBox(0, x.why().c_str(), "OLE Exception", MB_OK);
}
```

Der erste Parameter des Konstruktors *TRegistrar* ist die Programmregistrierstruktur, die vereinbarungsgemäß *AppReg* heißt. Der zweite Parameter ist eine Factory-Callback-Funktion. Das Beispiel erzeugt den Callback mittels eines Factory-Templates. Bei einem Automatisierungs-Server, der ObjectWindows nicht verwendet, benutzt man das Template *TOcAutoFactory. (Weitere Hinweise zu Factory-Callbacks und Templates finden sich im ObjectWindows Referenzhandbuch.*)

Der Aufruf von *IsOptionSet* ermittelt, ob dem Anwendung ein Kommandozeilenschalter übergeben wurde, der es veranlaßt, sich in der Registrierdatenbank des Systems anzumelden und dann zu stoppen; war dies nicht der Fall, so aktiviert das Programm *Run*. Die Registrierung ruft sodann den Factory-Callback auf, wo sich die Botschaftsschleife befindet. Mit der Abarbeitung von *Run* ist das Anwendung beendet.

Weitere Hinweise zu Kommandozeilenschaltern finden Sie in Tabelle 21.2.

Deklaration von automatisierbaren Methoden und Eigenschaften

Für die Automatisierung einer Klasse müssen zwei Tabellen erzeugt werden, eine in der Klassendeklaration und eine in der Klassenimplementation. Die erste Tabelle wird als *Automatisierungsdeklaration* bezeichnet und beschreibt, welche Elemente der Klasse einem Kontroller zugänglich sind. Die zweite Tabelle ist die *Automatsierungsdefinition* und definiert öffentliche Namen, über die ein Kontroller die verschiedenen zugänglichen Klassenelemente erreicht. Im folgenden wird erläutert, wie man eine Automatisierungsdeklaration erzeugt.

Die Automatisierungsdeklaration muß in der Deklaration einer automatisierten Klasse stehen. Sie beginnt mit dem Makro DECLARE_AUTOCLASS und beinhaltet einen Eintrag für jedes Klassenelement, das zugänglich gemacht werden soll. Die Makros fügen geschachtelte Klassen hinzu, die von ObjectComponents für die Verarbeitung der von OLE

kommenden Befehle instantiiert werden. Struktur und Größe der Ausgangsklasse werden nicht verändert.

Das folgende Beispiel einer Automatisierungsdeklaration macht Funktionen und Datenelemente einer C++ Klasse zugänglich, die einen Taschenrechner simuliert:

```
DECLARE_AUTOCLASS(TCalc)
  AUTODATA (Accum,  Accum,   long, )
  AUTODATA (Opnd,  Opnd,   long, )
  AUTODATA  (Op,      Op, short, AUTOVALIDATE(Val>=OP_NONE && Val<=OP_
LEAR))
  AUTOFUNC0 (Eval,    Eval,      TBool, )
  AUTOFUNC0V(Clear,   Clear,            )
  AUTOFUNC0V(Display, Display,          )
  AUTOFUNC0V(Quit,    Quit,             )
  AUTOFUNC1 (Button,  Button,    TBool, TAutoString,)
  AUTOFUNC0 (Window,  GetWindow, TAutoObject<TCalcWindow>,  )
  AUTOFUNC1 (LookAt,  LookAtWindow, long, TAutoObject<const TCalcWindow>,)
  AUTODATAR0(MyArray, Elem,      TAutoObjectByVal<TMyArray>,)
```

Die automatisierte Klasse hat die Bezeichnung TCalc. Jedes AUTOFUNC- bzw. *AUTODATA-Makro macht ein Element von TCalc* zugänglich. Zu den Funktionen des *TCalc*-Elementen zählen unter anderem *Eval*, *Clear*, *Display* und *Quit*. Zu seinen Datenelementen gehören *Accum*, *Opnd*, *Op* und *Elem*. *TCalc* hat noch weitere Elemente, die es nicht automatisiert und deshalb nicht in die Deklarationstabelle aufnimmt.

Für eine Automatisierungsdeklaration ist kein Abschlußmakro erforderlich; das Makro END_AUTOCLASS, mit dem man eine Automatisierungsdefinition abschließt, wird hier also nicht benutzt. Zu beachten ist außerdem, daß jede Zeile der Deklaration mit einer rechten Klammer und nicht mit einem Interpunktionszeichen endet.

Die Automatisierungsdeklaration sollte am Ende einer Klassendefinition stehen, weil die Makros den Zugriffsspezifizierer modifizieren könnten. Falls man die Deklaration an einer anderen Stelle angeben möchte, muß unmittelbar dahinter ein Zugriffsspezifizierer folgen (**public**, **protected** oder **private**).

Deklarationsmakros erstellen

Jedes Makro in einer Automatisierungsdeklaration beschreibt eine einzelne Methode oder Eigenschaft, die andere Programme manipulieren können, und stellt jeweils unterschiedliche Arten von Klassenelementen zur Verfügung. AUTOFUNC1 beispielsweise stellt eine Elementfunktion zur Verfügung, die einen einzelnen Parameter verarbeitet. AUTOFUNC2V stellt demgegenüber eine Funktion mit zwei Parametern zur Verfügung, die keinen Wert zurückgibt (**void**). AUTOPROP macht eine Eigenschaft über Set- und Get-Funktionen zugänglich, die einen Einzelwert einfügen

oder zurückliefern. AUTODATA stellt ein Datenelement bereit, das der Kontroller direkt lesen und modifizieren kann. Kapitel 6 im *ObjectWindows Referenzhandbuch* enthält eine komplette Tabelle der Makros zur Automatisierungsdeklaration.

Automatisierungsmakros haben folgendes allgemeine Format:

```
MAKRONAME( InternalName, FunctionName, ReturnType, ArgumentType,
           Options)
```

Nicht jedes Makro verwendet alle fünf Parameter. AUTOFUNC1V beispielsweise kommt ohne *ReturnType* aus, da er ein **void** zurückliefert. AUTOFUNC0 hat gar keine Argumente, während AUTOFUNC2 zwei verschiedene Argumente hat. Soweit Argumente jedoch benötigt werden, müssen sie in der vorgegebenen Reihenfolge erscheinen.

InternalName ist der Bezeichner, den Sie der jeweiligen automatisierbaren Eigenschaft oder Funktion zuweisen wird. ObjectComponents benutzt diesen Namen intern, um den Überblick über die verschiedenen Elemente zu behalten. Ansonsten erscheint der interne Name nur noch im zugehörigen Eintrag der Automatisierungsdefinitions-Tabelle der betreffenden Klasse. Der interne Name ist ein eindeutiger Bezeichner für das Element. (Im Quelltext verwendete Namen sind nicht unbedingt eindeutig; sie können beispielsweise überladen werden.)

FunctionName ist der Name, unter dem die betreffende Eigenschaft oder Funktion im Quelltext angesprochen wird, und kann ein beliebiger Ausdruck sein, der einen Funktionsaufruf ergibt; der Ausdruck muß allerdings innerhalb des Geltungsbereiches des automatisierten Objektes definiert sein. ObjectComponents versucht, die Funktion über den Zeiger **this** zu erreichen.

Interne Namen und Funktionsnamen sollten identisch sein, sofern nicht der Funktionsname überladen oder indirekt referenziert wird (Indirektion). Beispielsweise enthält eine Klasse ein Datenelement, das auf ein anderes Objekt zeigt:

```
TObject* MyObject;
```

Um nun einen Funktionsaufruf wie *MyObject->MyFunction* zugänglich zu machen, sollte man einen internen Namen angeben, der keine Indirektion verwendet; im vorliegenden Fall könnte das beispielsweise *MyFunction* sein:

```
AUTOFUNC0V( MyFunction, MyObject->MyFunction, )
```

Wenn eine Funktion überladen wird, verwendet man für alle Versionen der Funktion denselben Funktionsnamen, gibt aber jeder einen anderen internen Namen. ObjectComponents kann überladene Funktionen an-

hand der Art ihrer Rückgabewerte und der Argumenttypen voneinander unterscheiden.

Die Parameter *ReturnType* und *ArgumentType* können ein beliebiger C Grunddatenyp (wie **int** oder **char**) oder aber ein Zeiger auf einen solchen Grunddatentyp sein. Zwei Datentypen müssen auf besondere Weise deklariert werden, damit ObjectComponents Zeiger korrekt umwandeln kann. Ist der Datentyp ein String (Typ **char***), so muß man ihn als *TAutoString* umdefinieren. Ist er ein Zeiger oder eine Referenz auf ein C++ Objekt, so deklariert man ihn im Rahmen von *TAutoObject*<>. Im Gegensatz zu anderen Grunddatentypen enthält ein Zeiger keinen Hinweis auf den von ihm dargestellten Wert. *TAutoString* und *TAutoObject* ordnen dem Zeiger Typinformationen zu, so daß ObjectComponents die korrekten Informationen zwischen Server- und Kontrollerprogrammen übermitteln kann.

Solche Typersetzungen erleichtern ObjectComponents-Umwandlungen zwischen C++ Datentypen und dem von OLE verwendeten Union-Typ VARIANT. Zeiger und Objektreferenzen sind besonders schwierig umzuwandeln, weil sie auf nicht zur Variablen gehörende Daten verweisen. Die Klassen *TAutoString* und *TAutoObject* liefern somit nützliche Typinformationen für die Konvertierung.

Das Beispiel *TCalc* demonstriert den Gebrauch von *TAutoObject*. Eine der von *TCalc* zugänglich gemachten Funktionen ist *GetFunction*, die eine Referenz auf ein *TCalcWindow*-Objekt zurückliefert.

```
AUTOFUNC0 (Window, GetWindow, TAutoObject<TCalcWindow>, )
```

Durch die Deklaration des Ergebnistyps als *TCalcWindow* wird mittels des *TAutoObject*-Templates ein sich selbst beschreibender Smart-Zeiger auf ein *TCalcWindow*-Objekt erzeugt.

Installation von wahlfreien Hooks für Prüf- und Filterungszwecke

Der abschließende Parameter jedes Automatisierungsmakros benennt eine Hook-Funktion, die immer dann aufgerufen wird, wenn OLE das zugänglich gemachte Klassenelement aufruft. Der *Hook*-Code wird bei jeder Verwendung des betreffenden Klassenelementes ausgeführt. ObjectComponents unterstützt Hooks zur Befehlsaufzeichnung, zum Befehlswiderruf, zur Prüfung von Befehlsargumenten und zum Überschreiben der implementierten Form von Befehlen. Hooks sind immer wahlfrei.

Zur Installation eines Hooks gibt man eines der folgenden Makros als letzten Parameter einer Automatisierungsdeklaration an:

- AUTOINVOKE
- AUTONOHOOK
- AUTORECORD

- AUTOREPORT
- AUTOUNDO
- AUTOVALIDATE

Jedes Makro benötigt als Parameter den auszuführenden Code; das Format des gewünschten Makros hängt von seiner Funktion ab.

Für eine Argumentprüfung beispielsweise sollte der Code ein Boolescher Ausdruck sein. Das Datenelement *Op* von*TCalc* enthält eine Ganzzahl, die die durchzuführende Operation (z.B. Addition oder Subtraktion) identifiziert. Die Automatisierungsdeklaration stellt mittels eines Hooks sicher, daß der *Op* zugewiesene Wert innerhalb des gültigen Bereiches von Operatornamen liegt.

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && Val<=OP_CLEAR))
```

AUTOVALIDATE übergibt den für die Prüfung auszuführenden Ausdruck. Innerhalb des Prüfungsausdrucks repräsentiert der Name *Val* den vom Kontroller übermittelten Wert. Für die Prüfung von Funktionsargumenten benutzt AUTOVALIDATE die Namen *Arg1*, *Arg2*, *Arg3* usw.

Sobald ein Automatisierungs-Kontroller versucht, einen Wert im Datenelement *Op* zu setzen, prüft ObjectComponents, ob der neue Werte im Bereich OP_NONE bis OP_CLEAR liegt. Ist dies nicht der Fall, so annulliert ObjectComponents den Befehl und sendet ein Fehlerergebnis an OLE.

Der an AUTOVALIDATE übergebene Ausdruck kann Funktionsaufrufe enthalten.

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && NotTooBig(Val))
```

ObjectComponents ruft nun *NotTooBig* auf, sobald ein Kontroller versucht, *Op* zu ändern.

```
bool NotTooBig(int Val) {
return (Val <= OP_CLEAR)
}
```

Definition externer Methoden und Eigenschaften

Neben der Deklaration ihrer automatisierbaren Elemente muß eine automatisierte Klasse eine zweite Tabelle mit Makros erzeugen, die öffentliche Symbole für die Referenzierung von zugänglich gemachten Methoden und Eigenschaften festlegen. Diese öffentlichen Symbole sind für andere Programme sichtbar; sie bilden die Schnittstelle des Kontrollers zu einem automatisierten OLE-Objekt.

Intern verknüpft ObjectComponents die öffentlichen Namen mit dem oder den zur Implementierung des OLE-Objektes angelegten C++ Objekt(en). Die Automatisierungsdeklaration gibt an, welche Klassenelemente zugänglich zu machen sind, und die Automatisierungsdefinition weist ihnen Namen zu.

Die Automatisierungsdefinition ist Teil der Klassenimplementation. Sie beginnt mit dem Makro DEFINE_AUTOCLASS und endet mit END_AUTOCLASS. Die Automatisierungsdefinition für *TCalc* hat folgendes Aussehen:

```
   DEFINE_AUTOCLASS(TCalc)
  EXPOSE_PROPRW(Opnd,       TAutoLong,   "Operand",       "@Operand_", HC_TC
LC_OPERAND)
  EXPOSE_PROPRW_ID(0,Accum,TAutoLong,   "!Accumulator", "@Accumulator_",
        HC_TCALC_ACCUMULATOR)
  EXPOSE_PROPRW(Op,         CalcOps,     "Op",            "@Op_", HC_TCALC_OPERATOR)
  EXPOSE_METHOD(Eval,       TAutoBool,   "!Evaluate",
"@Evaluate_", HC_TCALC_EVALUATE)
  EXPOSE_METHOD(Clear,      TAutoVoid,   "!Clear",        "@Clear_", HC_TCALC_CLEAR)
  EXPOSE_METHOD(Display,    TAutoVoid,   "!Display",      "@Display_", HC_TCALC_DISPLAY)
  EXPOSE_METHOD(Quit,       TAutoVoid,   "!Quit",         "@Quit_", HC_TCALC_QUIT)
  EXPOSE_METHOD(Button,     TAutoBool,   "!Button",       "@Button_", HC_TCALC_BUTTON)
    REQUIRED_ARG(            TAutoString,"!Key")
  EXPOSE_PROPRO(Window,     TCalcWindow,"!Window",       "@Window_", HC_TCALC_WINDOW)
  EXPOSE_METHOD(LookAt,     TAutoLong,   "!LookAtWindow","@LookAtWindow_",
        HC_TCALC_LOOKATWINDOW)
    REQUIRED_ARG(            TCalcWindow,"!Window")
  EXPOSE_PROPRO(MyArray,    TMyArray,    "!Array",        "@Array_", HC_TCALC_ARRAY)
  EXPOSE_APPLICATION(       TCalc,       "!Application", "@Application_",
        HC_TCALC_APPLICATION)
END_AUTOCLASS(TCalc, "TCalc", "@TCalc", HC_TCALC)
```

Die Makros EXPOSE_*xxxx* weisen Namen für Methoden und Eigenschaften zu. EXPOSE_PROPRW definiert eine Eigenschaft, auf die Kontroller Lese- und Schreibzugriff haben. EXPOSE_PROPRO erlaubt dem Kontroller nur das Lesen des Eigenschaftswertes. REQUIRED_ARG weist einen Namen für ein Funktionsargument zu.

Beispielsweise aktiviert ein Kontroller die Funktion *LookAt* über den Namen *LookAtWindow* und ruft damit den den Parameter W*indow* der Funktion auf. END_CLASS teilt Objekten des Typs *TCalc* den öffentlichen Namen “Tcalc” zu.

Die meisten Strings in dieser Automatisierungsdefinition beginnen mit einem Symbol (! oder @). Diese Symbole zeigen an, daß die Anwendung *AutoCalc* in ihren Ressourcen über Übersetzungen für die öffentlichen Symbole verfügt. Jeder von einem Kontroller kommende Befehl enthält eine Locale-ID als Hinweis auf die vom Kontroller benutzte Sprache. Wurde der Kontroller beispielsweise in Deutsch geschrieben, so kann er anstelle des Strings "Evaluate" die Übersetzung "Auswerten" übergeben, die dann von ObjectComponents korrekt als Aufruf der Funktion *Eval* interpretiert wird. Weitere Informationen über die Verwendung internationaler Strings finden sich im Abschnitt “Lokalisierung von Symbolnamen” später in diesem Kapitel.

Jedes in der Automatisierungsdefinition beschriebene Element muß zuvor in der Automatisierungsdeklaration aufgeführt worden sein; dies bedeutet beispielsweise, daß für jeden mit EXPOSE_METHOD definierten Funktionsnamen eine entsprechende AUTOFUNC-Deklaration vorhanden sein muß. Für jedes EXPOSE_PROP muß es ein entsprechendes AUTOPROP, AUTOFUNC, AUTOFLAG oder AUTODATA geben, je nachdem wie man die Eigenschaft implementiert.

Definitionsmakros

Makros, die Methoden und Eigenschaften zugänglich machen, haben fünf Parameter: interner Name, Typ des zurückgelieferten Ergebnisses, externer Name und Dokumentationsstring. Mit dem wahlfreien fünften Parameter kann man einem Element einen Hilfekontext-ID zuweisen.

```
MAKRONAME(InternalName, ReturnType, ExternalName, DocString,
          HelpContext )
```

- *InternalName* ist der Name, der dem Element in der Automatisierungsdeklaration zugewiesen wurde.
- *ReturnType* gibt an, welche Art von Automatisierungsdaten die Methode zurückliefert bzw. die Eigenschaft enthält. Die möglichen Datentypen sind in der letzten Spalte von Tabelle 22.2 im vorliegenden Kapitel aufgeführt.
- *ExternalName* ist das, was der Automatisierungs-Kontroller sieht. Ein Benutzer, der Kommandos über einen Kontroller senden, kann alle Eigenschaften und Methoden über ihren externen Namen ansprechen.
- *DocString* soll dem Anwender Auskunft über den Zweck der betreffenden Eigenschaft oder Methode geben. OLE zeigt diesen String an, wenn der Anwender im Zusammenhang mit einem bestimmten Automatisierungsbefehl um Hilfe bittet. Soll kein Dokumentstring bereitgestellt werden, so setzt man den Parameter auf 0.
- Der fünfte Parameter *HelpContext* ist wahlfrei. Die hier angegebene Zahl spricht einen bestimmten Abschnitt einer Windows-Hilfedatei (.HLP) an. Bei Bedarf können Sie eine Hilfedatei mit Angaben zur Syntax und Verwendung aller zugänglich gemachten Elemente anlegen. Wenn Sie die Kontext-IDs für die einzelnen Elemente in der Automatisierungsdefinition der Klasse angeben, kann ein Automatisierungs-Kontroller von OLE die entsprechenden Hilfebildschirme für den Anwender anfordern. Ein Anwender beispielsweise, der ein Automatisierungsskript schreibt, kann dann zur Laufzeit eine Liste der von Ihrer Anwendung zugänglich gemachten Elemente suchen, sich die Dokumentstrings anzeigen lassen und sogar einen Hilfebildschirm für jedes Element abrufen.

Falls Sie eine Hilfedatei für die Automatisierung bereitstellen möchten, geben Sie den Dateinamen über den Schlüssel *typehelp* an (siehe Tabelle 22.1).

Die Tabelle der Makros für die Automatisierungsdefinition Kapitel 5 des *ObjectWindows Referenzhandbuch* enthält eine vollständige Liste der diversen EXPOSE-Makros.

Soll eine Methode zugänglich gemacht werden, die Argumente benötigt, so muß man zusätzlich zur Definition ein Makro bereitstellen, der die einzelnen Argumente beschreibt. Das folgende Muster gilt für eine Funktion mit drei Argumenten und zeigt neben der Definition auch die Makros, die die Methode für die Automatisierung definieren:

```
   // Deklaration der Elementfunktion
long TCalculator::AddNumbers(short Num1, short Num2 = 0, short Num3 = 0);

// Später erscheinen diese Angaben nach DEFINE_AUTOCLASS(TCalculator)
EXPOSE_METHOD(AddNumbers, TAutoLong, "AddNumbers", "Sum up to 3 numbers",
HC_ADDNUMBERS)
  REQUIRED_ARG(TAutoShort, "Num1")
  OPTIONAL_ARG(TAutoShort, "Num2", '0')
  OPTIONAL_ARG(TAutoShort, "Num3", "0")
```

Das erste Argument *Num1* ist obligatorisch; die beiden anderen sind wahlfrei. Alle drei Argumente sind vom Typ **short int**. Bei der Beschreibung von wahlfreien Argumenten muß ein Vorgabewert angegeben werden; im obigen Beispiel ist dies *0*.

Die Konventionen von OLE empfehlen, daß jedes Automatisierungsobjekt eine Eigenschaft enthält, die das zugehörige Anwendung repräsentiert; diese Eigenschaft kann man der Automatisierungsdefinition mit dem Makro EXPOSE_APPLICATION hinzufügen:

```
EXPOSE_APPLICATION(TMyClass, "Application", "My Application",)
```

In EXPOSE_APPLICATION muß dieselbe Klasse angegeben werden wie für die Factory-Template-Klasse (siehe "Erzeugung eines Registrarobjektes").

Datentypbezeichner in einer Automatisierungsdefinition

Die meisten Makros in einer Automatisierungsdefinition brauchen eine Datentypbeschreibung - den Typ des von der Funktion zurückgelieferten Wertes, den Typ der verschiedenen Funktionsargumente oder den Typ eines Datenelementes. Die möglichen Datentypwerte in einer Automatisierungsdefinition sind keine vordefinierten Datentypen von C und sind im folgenden angegeben

- Ein zuvor für eine Automatisiserung definierter Aufzählungswert (dieses Verfahren wird im nächsten Abschnitt erläutert)

- Der Name einer automatisierten Klasse (z.B. TCalc)
- Eine der vordefinierten Klassen, die ObjectComponents für die Darstellung von C-Grunddatentypen bereitstellt. Siehe hierzu Tabelle 22.2.

Die Verwendung von vordefinierten Klassen statt von vordefinierten Datentypen in C hat den Zweck, für Suchvorgänge vom Kontroller aus Typinformationen bereitzustellen. Für zugänglich gemachte Klassen kann ObjectComponents durch RTTI Typinformationen ermitteln. Die Automatisierungsdatentypen in der folgenden Tabelle sind als datenlose Strukturen definiert; sie ermitteln einfach einen statischen Wert, der einen Datentyp anzeigt. Die Bezeichnerwerte sind identisch mit denen, die OLE zur Unterscheidung der von ihm unterstützten Datentypen verwendet. Alle Automatisierungsdatentypen leiten sich von einer Basisklasse namens *TAutoVal*, d.h. sie sind polymorph. ObjectComponents kann in der Praxis von jedem über die Automatisierung übermittelten Wert eine Beschreibung seines eigenen Datentyps anfordern.

Tabelle 22.2 führt alle Automatisierungsdatentypen und den Zusammenhang auf, in dem sie benutzt werden. In der linken Spalte ermittelt man den Typ, den die automatisierte Klasse für ihre Argumenten oder Rückgabewerte verwendet. Die beiden anderen Spalten geben Auskunft darüber, welcher Datentyp in den entsprechenden Einträgen der Automatisierungsdeklaration und -definition anzugeben ist.

Tabelle 22.2: Automatisierungsdatentypen

C++ Typ (in Klasse)	Transfertyp (in DECLARE_AUTOCLASS)	Zugänglicher Typ (in DEFINE_AUTOCLASS)
short	short	TAutoShort
unsigned	short	short oder unsigned TAutoShort oder TAutoLong
long	long	TAutoShort
unsigned long	unsigned long	TAutoLong (wird als signed long behandelt)
int	int	TAutoInt
unsigned int	int oder long	TAutoInt oder TAutoLong
float	float	TAutoFloat
double	double	TAutoDouble
bool (oder int)	TBool	TAutoBool
TAutoDate	TAutoDate	TAutoDate
TAutoDate far *	TAutoDate far*	TAutoDateRef
TAutoCurrency far*	TAutoCurrency far*	TAutoCurrencyRef
char*	TAutoString	TAutoString
const char*	TAutoString	TAutoString
char far*	TAutoString	TAutoString
const char far*	TAutoString	TAutoString
string	string	TAutoString
enum	short oder int	TAutoShort oder user-defined AUTOENUM
T*	TAutoObject<T>	T (Klasse T muß automatisiert sein)
T&	TAutoObject<T>	T (Klasse T muß automatisiert sein)
const T*	TAutoObject<const T>	T (Klasse T muß automatisiert sein)

const T&	TAutoObject<const T>	T (Klasse T muß automatisiert sein)
T* (Rückgabe)	TAutoObjectDelete<T>	(C++ Objekt gelöscht, wenn keine Refs)
T& (Rückgabe)	TAutoObjectDelete<T>	(C++ Objekt gelöscht, wenn keine Refs)
T (Rückgabe)	TAutoObjectByVal<T>	T (T kopiert, gelöscht wenn refs==0)
void (keine Rückgabe)	(verwende AUTOFUNCxV-Makros	TAutoVoid
)short far*	short far*	TAutoShortRef
long far*	long far*	TAutoLongRef
float far*	float far*	TAutoFloatRef
double far*	double far*	TAutoDouble Ref

Zugänglichmachen von Daten zu Aufzählungszwecken

Ein Automatisierungs-Server muß unter Umständen auch Aufzählungswerte zugänglich machen. Wenn Sie eine Gruppe interner Datenwerte zugänglich machen und sie über lokalisierbare Strings ansprechen möchten, verwenden Sie dazu OLE-Aufzählungen. Beispielsweise legt *AutoCalc* Aufzählungsoperatoren fest, welche die verschiedenen Operationen darstellen, die der Taschenrechner mit Zahlen durchführen kann.

```
enum operators {
  OP_NONE = 0,
  OP_PLUS,
  OP_MINUS,
  OP_MULT,
  OP_DIV,
  OP_EQUALS,
  OP_CLEAR,
};
```

Sobald der Taschenrechner Eingaben erhält, speichert er die vorgesehene mathematische Operation in einem privaten Datenelement namens *Op*.

```
short Op;
```

Operationen werden über verschiedene Konstanten OP_*xxxx* definiert. Die Methode *Eval führt die vorgesehene Operation für die soeben eingegebene Zahl und den Akkumulatorinhalt des Taschenrechners durch. AutoCalc* macht das Datenelement *Op* für die Automatisierung verfügbar, so daß ein Kontroller Operatoren direkt eingeben kann. Die Automatisierungsdeklaration sieht so aus:

```
AUTODATA(Op, Op, short, AUTOVALIDATE(Val>=OP_NONE && Val<=OP_CLEAR))
```

Die Automatisierungsdeklaration zeigt, daß das Datenelement *Op* einen Short-Wert enthält. Die Symbole OP_PLUS und OP_MINUS sind jedoch nur innerhalb des Serverprogramms definiert; der Kontroller kann sie bei der Befehlsübergabe nicht benutzen. Im Idealfall sollte der Kontroller imstande sein, in Skripts allgemeinverständliche Strings wie "Add" oder "Substract" zu benutzen.

Öffentliche Symbole werden in der Automatisierungsdefinition deklariert. Leiten Sie eine Tabelle, die Symbole für die Aufzählungswerte definiert, mit dem Makro DEFINE_AUTOENUM ein.

```
 DEFINE_AUTOENUM(CalcOps, TAutoShort)
  AUTOENUM("Add",      OP_PLUS)
  AUTOENUM("Subtract", OP_MINUS)
  AUTOENUM("Multiply", OP_MULT)
  AUTOENUM("Divide",   OP_DIV)
  AUTOENUM("Equals",   OP_EQUALS)
  AUTOENUM("Clear",    OP_CLEAR)
END_AUTOENUM(CalcOps, TAutoShort)
```

Das Makro AUTOENUM verarbeitet zwei Parameter: einen Aufzählungsstring und eine Konstante. Der (lokalisierbare) Aufzählungsstring ist identisch mit dem externen Namen, der Kontrollern über OLE zugänglich gemacht wird.

Das einleitende und das abschließende Makro der Aufzählungstabelle weisen diesem Aufzählungstyp den Namen *CalcOps* und der Aufzählung den automatisierten Datenyp *TAutoShort* zu, da die Aufzählungswerte allesamt vom Typ **short int** sind.

Tabelle 22.3 zeigt die aufzählbaren C++ Typen sowie die entsprechenden automatisierbaren Typen, um sie zugänglich zu machen.

Tabelle 22.3: Aufzählbare C++ Typen und Automatisierungstypen, die sie zugänglich machen

C++ Typ	Aufzählungstyp (für DEFINE_AUTOENUM)
bool	TAutoBool
double	TAutoDouble
float	TAutoFloat
int	TAutoInt
long	TAutoLong
short	TAutoShort
const char*	TAutoString

Die Erzeugung einer Tabelle von Aufzählungswerten ergibt einen neuen Datentyp, mit dem sich Argumente und Rückgabewerte in einer Automatisierungsdefinition beschreiben lassen. Da ObjectComponents den Aufzählungstyp *CalcOps* versteht, kann man damit die Eigenschaft *Op* zu definieren.

```
EXPOSE_PROPRW(Op, CalcOps, "Op", "@Op_", HC_TCALC_OPERATOR)
```

Diese Zeile besagt, daß *Op* eine Lese/Schreib-Eigenschaft ist und Werte des Typs *CalcOps* enthält. Wenn der Kontroller versucht, ein “Multiply” oder “Divide” in die Eigenschaft von *Ops* zu schreiben, übersetzt ObjectComponents den String korrekt in den als OP_MULT oder OP_DIV definierten Wert.

Erzeugung des Servers

Um einen Automatisierungs-Server zu erzeugen, muß man die notwendigen Include-Header und die Verknüpfung zu den erforderlichen Bibliotheken einbinden.

Einbinden von Header-Dateien

Ein automatisiertes Programm muß folgende Header aufweisen:

```
    #include <ocf/automacr.h>   // Definitions- und Deklarationsmakros
  #include <ocf/ocreg.h>      // Klasse TRegistrar
```

Die Liste ist deshalb kurz, weil ein Automatisierungs-Server nicht sehr viele der für Verknüpfung und Einbettung verwendeten ObjectComponents-Klassen benötigt.

Kompilierung und Verknüpfung

Automatisierungs-Server und -Kontroller müssen mit dem Speichermodell *medium* oder *large* (das Modell *medium* ist schneller) kompiliert und mit den Bibliotheken von OLE und ObjectComponents verknüpft werden.

Wenn OLE-Unterstützung spezifiziert wird, wählt die IDE die richtigen Erzeugen-Optionen für Sie aus. Um ein ObjectComponents-Programm von der Kommandozeile aus zu erzeugen, erstellt man eine kurze Make-Datei, welche die im Unterverzeichnis von EXAMPLES stehende Datei OCFMAKE.GEN einbindet.

```
    EXERES = MYPROGRAM
 OBJEXE = winmain.obj myprogram.obj
 !include $(BCEXAMPLEDIR)\ocfmake.gen
```

EXERES und OBJRES enthalten den Namen der zu erzeugenden Datei sowie die Namen der zugrundeliegenden Objektdateien. Die letzte Zeile bindet die Datei OCFMAKE.GEN ein. Nennen Sie die Datei MAKEFILE, und geben Sie in der Kommandozeile folgendes ein:

```
make MODEL=l
```

Dadurch wird auf der Basis der Anweisungen in OCFMAKE.GEN eine neue, projektspezifische Make-Datei namens WIN16L*xx*.MAK erzeugt.

Weitere Informationen zu OCFMAKE.GEN und den für ein ObjectComponents-Programm benötigten Bibliotheken finden Sie im Abschnitt“Erzeugung eines ObjectComponents-Programms”.

Funktionsoptimierung des Automatisierungs-Servers

In den vorhergehenden Abschnitten wurde der zusätzliche Code beschrieben, den verschiedene Anwendungsarten aufweisen müssen, um

als Automatisierungs-Server zu fungieren. Die nachfolgenden Ausführungen erläutern, wie sich die Server-Leistung verbessern läßt.

Zusammenfassung mehrerer C++ Objekte zu einem einzigen OLE-Automatisierungsobjekt

Die komplette Menge der zu einem einzelnen automatisierbaren OLE-Objekt gehörenden Elementfunktionen und eigenschaften kann durch eine Kombination von C++ Objekten implementiert werden. Ein automatisierbarer Kalender beispielsweise kann mit einer TCalendar-Klasse beginnen. Unter Umständen muß aber das automatisierbare OLE-Kalender-Objekt Methoden und Eigenschaften zugänglich machen, über die das C++ Objekt *TCalendar* nicht verfügt. Beispielsweise könnte die Hintergrundfarbe von der Basisklasse von *TCalendar* vererbt werden, und vielleicht gehören einige Eingabefunktionen zu separaten Steuerfenstern im Client-Bereich des Kalenders. In einem solchen Fall muß die Automatisierungsdeklaration für *TCalendar* einige Aufgaben an andere C++ Klassen delegieren. Um mehrere C++ Objekte zu einem einzigen OLE-Objekt zusammenzufassen, fügt man entsprechende Makros in die Automatisierungsdefinition ein.

```
   // diese Zeilen müssen im einleitenden Definitionsblock von
DEFINE_AUTOCLASS(TCalendar) stehen
EXPOSE_INHERIT(TCalendarWindow, "CalendarWindow")
EXPOSE_DELEGATE(TWeekForwardButton, "WeekForward",
GetWeekForwardButton(this))
```

Eine zugänglich gemachte Klasse muß auch automatisiert sein, d.h. *TCalendarWindow* und *TWeekForwardButton* müssen ebenfalls ihre eigenen AUTOCLASS-Tabellen haben. Wenn man diese beiden Klassen in der Automatisierungsdefinition von *TCalendar* zugänglich macht, faßt man sämtliche zugänglich gemachten Elemente aller drei Klassen zu einer einzigen Symboltabelle zusammen. Sobald OLE einen Automatisierungsbefehl an den Kalender sendet, sucht ObjectComponents nach dem zugehörigen Klassenelement in *TCalendar*, dann in *TCalendarWindow* und schließlich in *TWeekForwardButton*.

Das Makro EXPOSE_DELEGATE hat als dritten Parameter eine Konvertierungsfunktion. Um Elemente in der Delegationsklasse erreichen zu können, benötigt ObjectComponents einen Zeiger auf ein Objekt dieser Klasse. Die Konvertierungsfunktion erhält einen Zeiger auf das aktuelle Objekt und liefert einen Zeiger auf das Delegationsobjekt zurück. Der Zeiger **this** wird dabei auf das andere Objekt gesetzt. Außerdem muß die Funktion global sein. Wenn beispielsweise ein Datenelement von *TCalendar* auf die Schaltfläche "Week Forward" zeigt, könnte die Konvertierungsfunktion so aussehen:

```
 TWeekForwardButton *GetWeekForwardButton ( TCalendar* this ) {
return( this->m_ForwardButton );
}
```

Soll eine vererbte Funktion oder Eigenschaft zugänglich gemacht werden, so ist keine Konvertierungsfunktion erforderlich; ObjectComponents kann in diesem Fall seine eigene Template-Konvertierungsfunktion für den Zugang zur Basisklasse erzeugen.

Hinweis

Es gibt noch eine andere Möglichkeit, die Aktionen mehrerer automatisierter Objekte innerhalb einer Anwendung zu koordinieren: man weist einem Objekt Zugriffsfunktionen zu, die die anderen Objekte zurückliefern. Beispielsweise automatisiert das Beispielprogramm *AutoCalc* fünf verschiedene Klassen, aber keine Klasse delegiert an eine andere. Wenn ein Kontroller ein Objekt vom *AutoCalc*-Server anfordert, erhält er lediglich das automatisierte Objekt *TCalc*. *TCalc* hat jedoch eine Eigenschaft namens *Window* zur Speicherung eines Objektes vom Typ *TCalcWindow*. *TCalcWindow* wiederum hat eine Eigenschaft, die die Kollektinon von Schaltflächen enthält. Das Kollektionsobjekt liefert einzelne Schaltflächen-Objekte zurück. Ohne Eigenschaften oder Funktionen, die die anderen Objekte zurückliefern, könnte der Kontroller nie darauf zugreifen. Achten Sie darauf bei Bedarf Zugriffsfunktionen hinzuzufügen!

Im Abschnitt "Übermittlung von Befehlen an die Kollektion" wird beschrieben, wie ein Kontroller Zugriffsfunktionen benutzt.

Mitteilung an OLE über gelöschtes Objekt

Wenn die Möglichkeit besteht, daß Ihr Programm sein automatisiertes Objekt löscht, während es noch mit einem Kontroller verbunden ist, muß OLE von dieser Löschung erfahren. Eine solche Vorsichtsmaßnahme ist nur von Bedeutung, wenn das Objekt aufgrund der Programmlogik während einer OLE-Sitzung durch nichtautomatisierte Mechanismen gelöscht werden könnte. Wenn OLE versucht, auf ein Automatisierungsobjekt zuzugreifen, dessen C++ Ausgangsobjekt gelöscht wurde, ist das gleichbedeutend mit dem Versuch, einen ungültigen Zeiger zu benutzen. Ein solcher Fehler kann durch einen Funktionsaufruf vermieden werden, der OLE über die Löschung des Objektes informiert.

```
    // Schreiben Sie diese Zeile in den Destruktor der automatisierten
    // Klasse
::GetAppDescriptor()->InvalidateObject(this);
```

Die globale Funktion *GetAppDescriptor* liefert einen Zeiger auf das Objekt *TAppDescriptor* der Anwendung zurück. *InvalidateObject* ist eine Methode von *TAppDescriptor*; sie teilt OLE mit, daß das an den Konstruktor des Deskriptors übergebene Objekt nun unzugänglich ist.

InvalidateObject sollte im Destruktor des Objekts aufgerufen werden, muß aber nicht. Sind Sie nicht Besitzer der Klasse, die zugänglich gemacht werden soll, so können Sie den Destruktor möglicherweise nicht ändern; dann ist auch die folgende Lösung möglich:

```
    TMyAutoClass MyAutomatedObject;

  ::GetAppDescriptor()->InvalidateObject(MyAutomatedObject);
  delete MyAutomatedObject;
```

Der an *InvalidateObject* übergebene Objektzeiger muß immer die am weitesten abgeleitete Form des Objektes ansprechen. Mit anderen Worten: ein polymorpher Zeiger muß auf die erzeugte Klasse, nicht auf eine ihrer Basisklassen weisen. Ein Aufruf von *InvalidateObject* im eigenen Destruktor des Objektes ist sicher, da *this* in diesem Fall immer auf die am weitesten abgeleitete Klasse zeigt. Wird *InvalidateObject* an anderer Stelle aufgerufen, so muß man gegebenenfalls mit der globalen Funktion *MostDerived* sicherstellen, daß das korrekte Objekt ungültig gemacht wird.

```
    appDesc->InvalidateObject(::MostDerived(MyPolymorphObject,
  typeid(MyPolymorphObject)));
```

In diesem Beispiel ist *MyPolymorphObject* ein Zeiger auf ein polymorphes Objekt und könnte somit auf eine Basisklasse oder auf ein von der Basis abgeleitetes Objekt beliebigen Typs weisen. *MostDerived* konvertiert den Zeiger: er weist nun auf ein Objekt desjenigen Typs, der am weitesten unten in der Hierarchie steht und deshalb die weiteste Ableitung von der Basis darstellt.

Neben dem *InvalidateObject*-Aufruf kann man OLE noch auf zwei weitere Arten darüber informieren, daß ein Objekt gelöscht wurde. Ein Verfahren besteht darin, die Objektklasse von *TAutoBase* abzuleiten. Der einzige Code in *TAutoBase* ist ein virtueller Destruktor, der *InvalidateObject* aufruft. Das folgende Beispiel deklariert eine Klasse namens *TMyAutoClass*. Wann immer ein Objekt des Typs*TMyAutoClass* gelöscht wird, weiß OLE darüber Bescheid.

```
    class TMyAutoClass: public TAutoBase { /* declarations */ };
```

Das zweite Verfahren besteht darin, daß man das Makro AUTODETACH in die Automatisierungsdeklarationstabelle der Klasse aufnimmt. In diesem Fall braucht die Klassenableitung nicht geändert zu werden; allerdings erhöht sich die Größe der Klasse um 1 Byte.

Lokalisierung von Symbolnamen

Symbole in einer Automatisierungsdefinition sind für andere OLE-Programme sichtbar, d.h. sie können in Skripts direkt angesprochen werden. Die Symbole werden Teil der Benutzeroberfläche des Programms. Für international eingesetzte Programme können die Strings in die jeweilige Landessprache übertragen werden; das amerikanische “Color” wird dann zu “Couleur” in einem französisischen, zu “Farbe” in einem deutschen und zu “Colour” in einem englischen Skript.

OLE unterstützt den Programmierer bei dieser Aufgabe; es übergibt einen Zahlenwert, der die vom Anwender gewählte Sprache bezeichnet.

Dieser Wert heißt Locale-ID oder LCID. LCIDs werden von OLE und Win32 API definiert und bestehen aus zwei Zahlen: die eine definiert die Sprache, die andere einen speziellen Dialekt dieser Sprache. Wenn OLE einen Automatisierungsaufruf an ein automatisiertes Programm übergibt, übergibt es gleichzeitig einen LCID. Der Automatisierungs-Kontroller kann den LCID zur Laufzeit aus den Systemeinstellungen ermitteln, oder aber die Spracheinstellung wird vom Benutzer des Kontrollers selbst eingestellt.

Von einem automatisierten Programm wird erwartet, daß es den LCID prüft und in die entsprechenden Übersetzungs-Strings umsetzt. Mit Hilfe von ObjectComponents kann der Programmierer eine Ressourcentabelle mit lokalisierten Versionen der benutzten Strings erzeugen. Bei der Bearbeitung von Automatisierungsaufrufen durchsucht ObjectComponents die Tabelle automatisch nach übersetzten Strings für die vom Kontroller angeforderte Sprache.

Zunächst wird dabei nach einem String mit der richtigen Sprache und DialektID gesucht. Ist kein Dialekt vorhanden, so wird die Suche fortgesetzt, diesmal aber nur nach der Sprache. Bleibt die Suche auch hier erfolglos, so verwendet ObjectComponents einfach den ursprünglichen, nicht übersetzten String.

Die Lokalisierung von Symbolen erfolgt in zwei Schritten.

1. Zunächst stellt man in den Programmressourcen Übersetzungen für die betreffenden Strings bereit.
2. Sodann kennzeichnet man im Quelltext die Strings, für die Übersetzungen vorhanden sind.

Eintragung von Übersetzungen in das Ressourcenskript

Eine Übersetzungstabelle in der Ressourcendatei (.RC-Datei) wird mit dem Ressourcentyp XLAT erzeugt.

```
#include "owl/locale.rh"

Left    XLAT FRENCH "Gauche" GERMAN "Links" SPANISH "Izquierda" XEND
Right   XLAT FRENCH "Droit"  GERMAN DUTCH "Rechts"              XEND
Center  XLAT ENGLISH_UK FRENCH GERMAN "Centre" SPANISH "Centro" XEND
Help    XLAT FRENCH "Aide" GERMAN "Hilfe" SPANISH "Ayuda"       XEND
```

Die Include-Datei *locale.rh* definiert XLAT als Ressourcentyp. XLAT und XEND sind Begrenzer für alle Übersetzungen eines gegebenen Strings. Derselbe Header definiert auch Makros zur Darstellung verschiedener Locale-IDs. FRENCH, DUTCH, und ENGLISH_UK repräsentieren beispielsweise jeweils einen unterschiedlichen LCID. UK ist ein Dialekt von ENGLISH.

Jede Zeile der Lokalisierungstabelle beginnt mit einem Ressourcenbezeichner. In den Beispielen wird zur Identifizierung der Ressource mit den Übersetzungen der Ausgangbegriff selbst benutzt.

Ein Lokalisierungstabelle braucht nicht für jeden String dieselbe Menge an Übersetzungen bereitzustellen. Es ist beispielsweise durchaus zulässig, für einen String die Übersetzungen FRENCH_FRANCE, FRENCH_BELGIUM und SWEDISH vorzusehen, für den nächsten String aber nur FRENCH und ITALIAN. Und wenn einmal derselbe String von mehreren Sprachen benutzt wird, braucht er, wie im nachfolgenden Beispiel, nur einmal definiert zu werden:

```
Center XLAT ENGLISH_UK FRENCH GERMAN "Centre" SPANISH "Centro" XEND
```

Im Englischen, Französischen und Deutschen wird hier das amerikanische “Center” mit “Centre” übersetzt, im Spanischen wird es zu “Centro.” Das Weglassen redundanter Übersetzungen macht die .EXE-Datei kleiner.

Kennzeichnung von übersetzbaren Strings im Quelltext

Mit der Erzeugung einer Ressourcentabelle ist der erste Schritt abgeschlossen; ObjectComponents muß aber noch wissen, wann es die Tabelle benutzen soll. Zu diesem Zweck kennzeichnet man in der Automatisierungsdefinition die übersetzbaren Strings mit einem vorangestellten Ausrufezeichen.

```
    EXPOSE_METHOD(Clear, TAutoVoid, "!Clear", "Clear accumulator",
  HC_TCALC_CLEAR)
```

Diese Zeile in *AutoCalc* macht eine Klassenmethode namens *Clear* zugänglich. *Clear* gibt **void** zurück. Der dritte Parameter *!Clear* spezifiziert den externen Namen, mit dem die Kontroller arbeiten. Das einleitende Ausrufezeichen weist ObjectComponents an, im ausführbaren Programm nach einer Lokalisierungsressource mit dem ID-String *Clear* zu suchen.

```
Clear XLAT GERMAN "AllesLöschen" XEND
```

Gleichzeitig kennzeichnet das Ausrufezeichen *Clear* als die sprachneutrale Form des Strings. Möchte ein Automatisierungs-Kontroller die Locale-ID GERMAN benutzen, so erhält er von ObjectComponents die Auskunft, daß die zugänglich gemachte Eigenschaft *AllesLöschen* heißt. Falls der Kontroller einen anderen Locale-ID verwendet, erhält er die sprachneutrale Form *Clear*.z

Argumentnamen lassen sich ebenso lokalisieren wie Eigenschaften und Methoden.

```
    EXPOSE_METHOD(Button, TAutoBool, "!Button", "Button push sequence",
  HC_TCALC_BUTTON)
    REQUIRED_ARG( TAutoString, "!Key")
```

Um festzustellen, wie die *Button*-Methode und ihr Argument genannt werden sollen, durchsucht ObjectComponents die Lokalisierungsressourcen des Programms für *Button* und *Key*.

```
    Button       XLAT GERMAN "Schaltfläche" XEND
Key          XLAT GERMAN "Taste" XEND
```

Der Suchalgorithmus unterscheidet nicht zwischen Groß- und Kleinschreibung; außerdem erlaubt die gegenwärtige Implementierung von 16-Bit-Windows in Ressourcennamen keine Verwendung von Zeichen des erweiterten Zeichenvorrates (z.B. diakritische Zeichen). Die beiden obigen Einschränkungen gelten jedoch nicht für die innerhalb einer Ressource gespeicherten Zeichen!

Ein Problem bei der Ressourcenbenennung tritt auf, wenn der String Leerstellen enthält - dies ist bei Ressourcennamen nicht zulässig! Sehen wir uns an einem Beispiel an, was bei der Lokalisierung des Beschreibungsstrings für folgende Eigenschaft geschieht:

```
    // unzulässig: Leerstellen sind in Ressourcenamen nicht erlaubt
EXPOSE_PROPRW(Caption, TAutoString, "!Caption", "!Window Title",
HC_TCALCWINDOW_TITLE)
```

Zwar ist es durchaus sinnvoll, Beschreibungen ebenso zu lokalisieren wie Eigenschaftsnamen, aber “Window title” ist kein zulässiger Ressourcenbezeichner. In solchen Fällen verwendet man als Einleitungszeichen ein @ anstelle eines ! und gibt dahinter einen zulässigen Bezeichner an.

```
    EXPOSE_PROPRW(Caption, TAutoString, "!Caption",
"@Caption_",HC_TCALCWINDOW_TITLE)
```

Das Präfix @ teilt ObjectComponents mit, daß der String *nur* ein Ressourcenbezeichner ist und nie angezeigt werden darf, ganz gleich welchen Locale-ID der Kontroller anfordert. Um diesen Unterschied für einen Leser des Programms noch klarer zu machen, müssen Strings, die ausschließlich als Bezeichner benutzt werden, mit einem Unterstrich abgeschlossen werden (wie in *Caption_*).

Um “Window Title” zum sprachneutralen String zu machen, ordnet man ihm keine Locale-ID in der Lokalisierungsressource zu.

```
    Caption_ XLAT "Window Title" GERMAN "Fenster-Aufschrift" XEND
```

Damit erhält ein Kontroller, der eine andere Locale-Einstellung als GERMAN anfordert, den String *Window Title*.

Neben ! und @ gibt es noch ein drittes Lokalisierungspräfix: #. Es leitet eine Ziffernfolge ein, über die eine Lokalisierungsressource angesprochen werden kann.

```
    EXPOSE_PROPRW(Caption, TAutoString, "!Caption",
"#10047",HC_TCALCWINDOW_TITLE)
```

Das Beispiel weist ObjectComponents an, nach einer Ressource mit der Nummer 10047 zu suchen. In der .RC-Datei muß die Ressource so erscheinen:

```
10047 XLAT "Window Title" GERMAN "Fenster-Aufschrift" XEND
```

Verwendung von XLAT-Ressourcen in ObjectComponents

Die externen Namen in Makros wie EXPOSE_METHOD und EXPOSE_PROPRW sind in Objekte des Typs *TLocaleString* eingebunden, ein lokalisierbares Substitut für Strings vom Typ **char***. Ein *TLocaleString*-Objekt enthält Befehle, die eine ausführbare Programmdatei nach XLAT-Ressourcen durchsuchen. Jeder Zugriff auf die Ressourcen von XLAT erfolgt über *TLocaleString*.

Die Klasse *TLocaleString* ist in *osl/locale.h* definiert. *TLocaleString* braucht nicht direkt angesprochen zu werden; sie wird von den Makros und Headers geholt.

TLocaleString ist sehr leistungsfähig. Wenn der Kontroller in der Eigensprache des Servers arbeitet, erkennt *TLocaleString*, daß die Strings im Quelltext bereits mit dem Locale übereinstimmen, und verzichtet auf das Durchsuchen von Ressourcen.

Normalerweise bestimmt ObjectComponents die Vorgabesprache der Anwendung, indem es den Locale-ID des Systems bei der Kompilierung liest und ihn im kompilierten Programm speichert. Der Programmierer kann den Vorgabewert aber durch eine Zeile der folgenden Art im Quelltext ändern:

```
   #include "olenls.h"
TLangId
TLocaleString::NativeLangId=MAKELANGID(LANG_ENGLISH,SUBLANG_ENGLISH_US);
```

Die Include-Datei olenls.h enthält Konstanten für die Unterstützung von Landessprachen, einschließlich des Makros *MAKELANGID und der Symbole für Sprache und Dialekt.*

Wenn *TLocaleString* auf Ressourcen zurückgreifen muß, versucht es, den Suchaufwand für Übersetzungen möglichst gering zu halten. Sobald es einen entsprechenden String für das aktuelle Locale gefunden hat, legt es ihn im Hauptspeicher ab und braucht ihn danach nie wieder zu laden; somit ist nur der erste Suchvorgang zeitaufwendig. Nachfolgenge Suchvorgänge werden schnell ausgeführt. Im Hauptspeicher werden die Strings in einer Hash-Tabelle gespeichert, um Duplikate zu vermeiden. Findet *TLocaleString* einen angeforderten String nicht, so merkt es sich dies und unternimmt später keinen weiteren Versuch mehr, ihn zu finden.

Lokalisierung von Registrierstrings

Derselbe Lokalisierungsmechanismus gilt für die von Ihrem Programm registrierten Strings. Einige Strings, wie z.B. *progid*, dürfen nicht lokalisiert werden; für folgende Registrierschlüssel ist jedoch eine Lokalisierung zulässig:

- debugdesc
- menuname
- typehelp
- description
- permname
- verb*n*
- *formatn*

Der folgende Auszug aus den Registriertabellen von *AutoCalc* zeigt, wo die Lokalisierungspräfixe anzugeben sind. Die Schlüssel *description* und *typehelp* werden lokalisiert.

```
  BEGIN_REGISTRATION(AppReg)
  REGDATA(clsid,       "{877B6200-7627-101B-B87C-0000C057CE4E}")
  REGDATA(progid,      APP_NAME ".Application")
  REGDATA(description,"@AppName")
  REGDATA(cmdline,     "/Automation")
  REGDATA(typehelp,    "@typehelp")
  REGDATA(version,     "1.2")
END_REGISTRATION
```

Weitere Hinweise zu den verschiedenen Registrierschlüsseln finden sich unter dem entsprechenden Namen im *ObjectWindows Referenzhandbuch*.

AutoCalc stellt in seinem Ressourcenskript Übersetzunen für die Strings *description* und *typehelp* bereit.

```
   AppName      XLAT "Automated Calculator 1.0 Application"
          GERMAN "Automatisierte Taschenrechner-Anwendung 1.0"
            XEND
typehelp     XLAT "autocalc.hlp"
          GERMAN "acalcger.hlp"
            XEND
```

ObjectComponents bestimmt die korrekte Sprache für die Registrierung zu Laufzeit anhand der Systemeinstellung; der Programmierer kann den Vorgabewert aber mit dem Language-Schalter ändern (siehe Tabelle 20.2).

Zugänglichmachen von Objektkollektionen

ObjectComponents läßt es zu, daß ein automatisiertes Objekt Kollektionen verschiedener Typen als Objekteigenschaften zugänglich macht. Die Elemente in einer Kollektion können zu einem Array, einer verketteten Liste oder einer anderen Struktur gehören, in der Mengen von gleicharti-

gen Elementen organisiert sind. Um eine Kollektion zugänglich zu machen, muß man Methoden zu ihrer Bearbeitung zugänglich machen. Zu diesen Methoden gehört in der Regel ein Zähler, der die Größe der Kollektion anzeigt, ein Iterator, um die Kollektion abzuarbeiten, und eine Direktzugriffsfunktion zum gezielten Aufgreifen einzelner Elemente der Kollektion.

Ein Kollektionsobjekt ist ein Objekt, das auf Anforderung einzelne Elemente aus einer Menge von zusammengehörigen Elementen zurückgibt; es implementiert die Methoden zur Bearbeitung von solchen Elementen. Im *AutoCalc*-Beispielprogramm sind die Schaltflächen auf dem Taschenrechner eine Menge von zusammengehörigen gleichartigen Objekten. *AutoCalc* definiert eine neue Klasse *TCalcButtons*, mit deren Methoden ein Kontroller einzelne Schaltflächenobjekte anfordern kann. Die Tasten selbst sind automatisierte Objekte; wenn also ein Kontroller eine Taste erhält, kann er einen Push-Befehl erteilen oder den von der Taste angezeigten Text ändern.

Um Kollektionen für eine Automatisierung zugänglich zu machen, sind im allgemeinen drei Schritte nötig.

- Man macht die Kollektion als Eigenschaft der übergeordneten Klasse zugänglich. Wie dies geschieht, hängt vom Konstruktor der Klasse ab, von der die Kollektion verwaltet wird.
- Man implementiert einen Iterator in der Kollektionsklasse.
- Man implementiert weitere Methoden in der Kollektionsklasse.

Erzeugen und Zugänglichmachen einer Kollektionsklasse

Bei der Konvertierung eines Anwendung, das Automatisierung unterstützen soll, wird sich vermutlich herausstellen, daß das Programm noch keine C++ Klasse aufweist, die als Kollektionsobjekt fungieren kann. Die Elemente können einfache Werte, Strukturen oder sogar durch Handles repräsentierte Systemobjekte sein. Also müssen Sie eine neue C++ Klasse erzeugen und sie in den Automatisierungstabellen der übergeordneten (Parent-) Klasse als Eigenschaft dieser Klasse zugänglich machen. Wie Sie die Kollektion in der Automatisierungsdeklaration der übergeordneten Klasse zugänglich machen, hängt von den Konstruktionsinformationen ab, die die übergeordnete Klasse dem Kollektionsobjekt zur Verfügung stellt. Die nachfolgenden Abschnitte behandeln einige mögliche Konstruktoren und zeigen die Makros, mit denen man die Kollektion den Eigenschaften der übergeordneten Klasse hinzufügen kann.

Instanzen der Kollektionsklasse werden nur auf Anforderung eines Kontrollers hin konstruiert. Das Kollektionsobjekt präsentiert sich gegenüber dem Kontroller als eine Eigenschaft der übergeordneten Klasse. Wenn beispielsweise in *AutoCalc* ein Kontroller den Inhalt der Eigenschaft von *Buttons* anfordert, erzeugt ObjectComponents sofort ein Objekt *TCalcBut-*

tons. Der Konstruktor eines Kollektionsobjektes muß ein einzelnes Argument verarbeiten, das ihm von der übergeordneten Klasse zur Initialisierung übergeben wird.

Da *TCalcButtons* eine Kollektion von untergeordneten Fenstern verwaltet, übergibt seine übergeordnete Klasse den Handle des übergeordneten Fensters. Der Konstruktor hat folgende Form:

```
TCalcButtons(HWND window) : HWnd(window) {}
```

Damit der Handle an den Konstruktor übergeben werden kann, muß die übergeordnete Klasse ihrer Automatisierungsdeklaration eine Zeile hinzufügen:

// aus der Automatisierungsdeklaration der übergeordneten Klasse

```
DECLARE_AUTOCLASS(TCalcWindow)
  AUTODATARO(Buttons, hWnd, TAutoObjectByVal<TCalcButtons>,)
```

Buttons wird als interner Name einer schreibgeschützten Eigenschaft mit dem Wert *TCalcWindow::hWnd* zugewiesen. Für den Datentyp dieser Eigenschaft gibt die Tabelle eine neue, auf der Kollektionsklasse basierende Klasse an. *TAutoObjectByVal<T>* bewirkt die Erzeugung einer Instanz von *T*, die so lange besteht, bis alle externen Referenzen auf sie freigegeben sind (d.h. wenn das zugänglich gemachte Objekt im Automatisierungs-Kontroller den Geltungsbereich verläßt).

TCalcWindow muß die Kollektionseigenschaft außerdem in der Automatisierungsdefinition zugänglich machen:

```
    // aus der Automatisierungsdefinition der übergeordneten Klasse
DEFINE_AUTOCLASS(TCalcWindow)
  EXPOSE_PROPRO(Buttons, TCalcButtons, "!Buttons", "@Buttons_",
HC_TCALCWINDOW_BUTTONS)
```

Wenn ein Kontroller den Inhalt der lesegeschützten Eigenschaft von *Buttons* anfordert, erzeugt ObjectComponents ein *TCalcButtons*-Objekt und übergibt seinem Konstruktor das Handle *hWnd*.

Die drei folgenden Beispiele zeigen weitere Möglichkeiten, wie eine übergeordnete Klasse ein Kollektionsobjekt als eine ihrer Eigenschaften bereitstellen kann:

- Fall 1**:** *TParent::DocList* zeigt auf den Anfang einer verketteten Liste von *TDocument*-Objekten. Als Kollektionsobjekt wird eine neue Klasse *TDocumentList* erzeugt. Der Konstruktor von *TDocumentList* erhält von seiner übergeordneten Klasse den Anfang der verketteten Liste:

  ```
  TDocumentList(TDocument*);
  ```

Die Automatisierungsdeklaration von *TParent* macht *DocList* als schreibgeschützte Eigenschaft zugänglich und weist ihr mittels der Kollektionsklasse einen Typ zu.

```
DECLARE_AUTOCLASS(TParent)
AUTODATARO(Documents, DocList, TAutoObjectByVal<TDocumentList>,)
```

Die Automatisierungsdefinition von *TParent* ruft die Kollektion *Documents* auf und gibt als ihren Typ *TDocumentList* an.

```
DEFINE_AUTOCLASS(TParent)
EXPOSE_PROPRO(Documents, TDocumentList, "Documents", "Doc Collection",
270)
```

- Fall 2**:** *TParent* enthält eine Liste. Es übergibt **this** an das Kollektionsobjekt *TList,* das die Listenelement mittels indirekter Referenzierung über den Zeiger der übergeordneten Klasse aufgreift. Der Konstruktor erhält den Zeiger.

  ```
  TList(TParent* owner)
  ```

  Die Automatisierungsdeklaration von *TParent* macht **this** als schreibgeschützte Eigenschaft zugänglich und weist ihr mittels der Kollektionsklasse einen Typ zu.

  ```
  DECLARE_AUTOCLASS(TParent)
  AUTOTHIS(List, TAutoObjectByVal<TList>,)
  ```

  Die Automatisierungsdefinition von *TParent* ruft die Kollektion *List* auf und gibt als ihren Typ *TList* an.

  ```
  DEFINE_AUTOCLASS(TParent)
  EXPOSE_PROPRO(List, TList, "List", "List of items", 240)
  ```

- Fall 3: *Elem* ist ein als **short** *Elem[COUNT]* definiertes Ganzahlen-Array. Das Kollektionsobjekt ist *TMyArray,* und der Konstruktor erhält von der übergeordneten Klasse einen Zeiger auf *Elem.*

  ```
  TMyArray(short* array)
  ```

  Die Automatisierungsdeklaration von *TParent* macht *Elem* als schreibgeschützte Eigenschaft zugänglich und weist ihr mittels der Kollektionsklasse einen Typ zu.

  ```
  DECLARE_AUTOCLASS(TParent)
  AUTODATARO(MyArray, Elem, TAutoObjectByVal<TMyArray>,)
  ```

  Die Automatisierungsdefinition von *TParent* ruft die Kollektion *Array* auf und gibt als ihren Typ *TMyArray* an.

  ```
  DEFINE_AUTOCLASS(TParent)
  EXPOSE_PROPRO(MyArray, TMyArray, "Array", "Array as collection", 110)
  ```

Implementierung eines Iterators für die Kollektion

Die Kollektionsklasse führt die Maßnahmen aus, die der Kontroller gemäß Ihren Vorgaben mit der Kollektion durchführen kann. Zu den Standard-Kollektionsmethoden gehören *Count* und *GetObject*, die die Anzahl der Elemente in der Kollektion oder einzelne über eine Nummer angesprochene Elemente zurückliefern. Die einzigen Methoden, die Sie selbst implementieren müssen, sind der Konstruktor und ein Iterator. Den Konstruktor haben Sie bereits gesehen. Eine Iteratorfunktion arbeitet die Kollektion ab und liefert bei jedem neuen Aufruf aufeinanderfolgende Elemente zurück.

Am einfachsten definiert man einen Iterator mit dem AUTOITERATOR-Makro, den man der Deklarationstabelle des Kollektionsobjektes hinzufügt.

```
DECLARE_AUTOCLASS(TCalcButtons)
AUTOITERATOR(int Id, Id = IDC_FIRSTID+1, Id <= IDC_LASTID, Id++,
             TAutoObjectByVal<TCalcButton>(::GetDlgItem(This->HWnd,Id)))
```

Die Parameter für AUTOITERATOR definieren den Algorithmus zur Aufzählung von Objekten in der Kollektion. Jedes der fünf Makroargumente stellt ein Programmfragment da, angeordnet wie in einer **for**-Schleife.

1. Deklarieren Sie Statusvariablen, mit denen Sie die Iterationen der Schleife verfolgen können, z.B.:

   ```
   int Index;
   ```

2. Weisen Sie den Statusvariablen Anfangswerte zu, z.B.:

   ```
   Index = 0;
   ```

3. Prüfen Sie mit einem Booleschen Ausdruck, ob die Schleife durchlaufen werden soll, z.B.:

   ```
   Index < This->Total
   ```

4. Ändern Sie Statusvariablen als Vorbereitung für die nächste Iteration, z.B.:

   ```
   Index++;
   ```

5. Holen Sie ein Element aus der Kollektion, z.B.:

   ```
   (This->Array)[Index];
   ```

Denken Sie daran, daß der Server für die Elemente jeden beliebigen Datentyp zurückliefern kann – Werte oder Objekte.

Im AUTOITERATOR-Parameter dürfen außer in Klammern keine Kommata verwendet werden. Semikolons dienen ausschließlich zur Trennung mehrerer Anweisungen, nicht jedoch als Abschluß eines Makroargumentes. Wie bei automatisierten Methoden ist *This* als der **this**-Zeiger der einschließenden C++ Klasse definiert (hier ist dies die Kollektion selbst).

AUTOITERATOR plaziert einen Iterator in die Tabelle der Automatisierungsdeklarationen. Das Iteratorelement muß in der Definitionstabelle zugänglich gemacht werden; dies geschieht mit dem EXPOSE_ITERATOR-Makro.

```
EXPOSE_ITERATOR(TAutoShort, "Array Iterator", HC_ARRAY_ITERATOR)
```

EXPOSE_ITERATOR benötigt weniger Parameter als andere Makros vom Typ EXPOSE_*xxxx*. Es werden keine internen oder externen Namen angegeben. Eine Klasse kann nur einen Iterator haben; der externe Name lautet immer *_NewEnum*. Der erste Parameter beschreibt den Typ der vom Iterator zurückgelieferten Elemente.

Der Automatisierungstyp beschreibt – wie eine Funktionsrückgabe – den Typ der vom Iterator zurückgelieferten Elemente. Das vorhergehende Beispiel zeigt die Iterierung eines Arrays von **short int**-Werten, d.h. der Automatisierungsdatentyp ist *TAutoShort*. (Eine Liste aller Automatisierungsdatentypen finden Sie in Tabelle 22.2.) Der zweite Parameter ist der Dokumentationsstring mit der Beschreibung der Iteratoreigenschaft, und der wahlfreie dritte Parameter bezeichnet einen .HLP-Dateikontext mit Informationen zum Iterator.

Hinweis

Von außen sieht ein Skriptkontroller den Enumerator als eine Eigenschaft mit dem reservierten Namen *_NewEnum*, die ein Objekt zurückliefert, welches die standardmäßige OLE-Schnittstelle *IENUMVariant* unterstützt. Diese Schnittstelle enthält Methoden zur Durchführung von Iterationen. Ein Kontroller verwendet einen Iterator in einer Schleife wie der folgenden, die mit Visual Basic für Anwendungen erstellt ist:

For Each Thing In Owner.Bunch ("Thing" ist ein beliebiger Iteratorname)

```
  Thing.Member.....          (kann auf Methoden und Eigenschaften
zugreifen)
  Next Thing                 (durchläuft alle Elemente der Kollektion)
```

Hinweis

Der AUTOITERATOR-Makro erzeugt innerhalb der Kollektionsklasse eine geschachtelte Klassendefinition. Komplizierte Iteratoren kann man auch ausdrücklich in C++ programmieren, wie das folgende Beispiel zeigt:

```
class TIterator : public TAutoIterator {
  public:
    ThisClass* This;
    /* Statusvariablen werden hier als Elementfunktionen deklariert */
    void Init() {/* Schleifeninitialisierung */}
    bool Test() {/* Abfrage der Schleifenbedingung*/}
    void Step() {/* Iteration;}
    void Return(TAutoVal& v) {/* aktuell zurückgemeldetes Objekt: v =
expr */}
    TIterator* Copy() {return new TIterator(*this);}
```

```
    TIterator(ThisClass* obj, TServedObject& owner)
                : This(obj), TAutoIterator(owner) {}
    static TAutoIterator* Build(ObjectPtr obj, TServedObject& owner)
    { return new TIterator((ThisClass*)obj, owner); }
};
friend class Titerator;
```

Hinzufügung weiterer Elemente zur Kollektionsklasse

Eine Kollektionsklasse macht nicht nur einen Iterator zugänglich, sondern gemäß den C++ Konventionen auch eine *Count*-Methode, die die Anzahl der Elemente in der Kollektion meldet, eine *Index*-Methode für den Direktzugriff auf Kollektionselemente und bei Bedarf auch Methoden wie *Add* und *Delete* zur externen Verwaltung der Kollektion. Als Beispiel finden Sie hier den kompletten Code für die Kollektionsklasse *TCalcButtons* in *AutoCalc*:

```
class TCalcButtons {    // Klasse wird nur temporär benutzt, um
Kollektion zugänglich zu machen
  public:
    TCalcButtons(HWND window) : HWnd(window) {}
    short GetCount() { return IDC_LASTID - IDC_FIRSTID; }
    HWND GetButton(short i) {return ::GetDlgItem(HWnd, i +
IDC_FIRSTID+1);}
    HWND HWnd;
  DECLARE_AUTOCLASS(TCalcButtons)
    AUTOFUNC0 (Count, GetCount, short,)
    AUTOFUNC1 (Item,  GetButton, TAutoObjectByVal<TCalcButton>, short,
                        AUTOVALIDATE(Arg1 >= 0 && Arg1 <
This->GetCount()) )
    AUTOITERATOR(int Id, Id = IDC_FIRSTID+1, Id <= IDC_LASTID, Id++,

TAutoObjectByVal<TCalcButton>(::GetDlgItem(This->HWnd,Id)))
};

DEFINE_AUTOCLASS(TCalcButtons)
  EXPOSE_PROPRO(Count, TAutoLong, "!Count", "@CountBu_",
HC_TCALCBUTTONS_COUNT)
  EXPOSE_ITERATOR(TCalcButton, "Button Iterator",
HC_TCALCBUTTONS_ITERATOR)
  EXPOSE_METHOD_ID(0, Item, TCalcButton,"!Item",  "@ItemBu_", 0)
    REQUIRED_ARG(TAutoShort, "!Index")
END_AUTOCLASS(TCalcButtons, "TButtonList", "@TCalcButtons",
HC_TCALCBUTTONS)
```

Erzeugung einer Typbibliothek

Eine Typbibliothek ist eine Binärdatei mit Informationen über einen Automatisierungs-Server. Die darin enthaltenen Informationen beschreiben die vom Server unterstützten Objekte, Eigenschaften und Methoden und werden von Programmier-Tools wie z.B. Automatisierungs-Kontrollern benutzt, die den Server aufrufen. Kontroller können in der Typbibliothek

Dokumentation und Hilfe zu spezifischen Objekten suchen. Der Pfad zur Typbibliothek ist eine der Informationen, die ein Automatisierungs-Server in der Registrierdatenbank des Systems ablegt.

ObjectComponents kann anhand der Angaben in den Automatisierungsdefinitionen des Servers eine Typbibliothek erzeugen. Rufen Sie dazu den Server mit dem Kommandozeilenschalter TypeLib auf.

```
myapp -TypeLib
```

Dieser Befehl veranlaßt ObjectComponents zur Erzeugung einer neuen Datei namens MYAPP.OLB im selben Verzeichnis wie MYAPP.EXE. ObjectComponents vermerkt den Pfad zur Bibliothek automatisch in der Registrierdatenbank.

Zusammen mit dem Schalter TypeLib kann man auch einen Pfad- und Dateinamen angeben.

```
myapp -TypeLib = data\mytyplib
```

ObjectComponents plaziert MYTYPLIB.OLB in ein Unterverzeichnis namens DATA unterhalb des Verzeichnis, in dem MYAPP.EXE steht.

Der SchalterLangID veranlaßt ObjectComponents, mehrere Typbibliotheken in unterschiedlichen Sprache anzulegen. Der folgende Befehl erzeugt zwei Typbibliotheken in Deutsch und Italienisch.

```
myapp -LangID=10 -TypeLib=italiano -LangID=7 -TypeLib=deutsch
```

Der Schalter LangID erfordert die Angabe einer Hexadezimalzahl. Win32 API definiert *80C* als Locale-ID für den belgischen Ableger des Französischen (d.h. Wallonisch). Damit der Befehl die gewünschte Wirkung hat, muß *myapp* natürlich in seinen XLAT-Ressourcen wallonische Strings bereithalten.

Weitere Hinweise zur Lokalisierung finden Sie im Abschnitt “Lokalisierung von Symbolnamen”. Der Abschnitt “Verarbeitung der Kommandozeile” enthält Informationen zu Befehlszeilenschaltern.

Die Typbibliothek kann auch um eine Online-Hilfe (HLP-Datei) ergänzt werden; diese Datei dokumentiert alle vom Server zugänglich gemachten Befehle sowie ihre Argumente und Anwendung. Sofern Sie über eine solche Hilfedatei verfügen, können Sie sie mit den Registrierschlüsseln *typehelp* und *helpdir* (siehe *ObjectWindows Referenzhandbuch*) registrieren. Der letzte Parameter des Makros EXPOSE_*xxxx in der Tabelle der Automatisierungsdefinitions-Tabelle ordnet jedem Befehl einen Hilfekontext-ID zu. Sobald der Automatisierungs-Kontroller Hilfe für einen Befehl anfordert, startet OLE die Hilfedatei automatisch.*

A N H A N G A

Konvertieren von ObjectWindows Code

ObjectWindows 2.5 ist eine mächtige neue Implementierung der Klassenbibliothek von ObjectWindows. Diese Version liefert eine Vielzahl von Merkmalen, die von den Benutzern von ObjectWindows 1.0 gefordert wurden:

- Größere Typensicherheit
- Verträglichkeit mit ANSI C++
- Unterstützung für mehrfache Vererbung
- Automatisiertes Umwandeln von Botschaften
- Breitere Kapselung der Windows-API, einschließlich der Unterstützung für GDI
- Mehrere neue Objekte auf hoher Stufe, einschließlich der Kapselungen von Mauspaletten- und Statuszeilenfunktionalität
- Transparentes Anvisieren von 16-Bit- und 32-Bit-Anwendungen für Windows NT, Win32s und Windows 3.1 von einer einzigen Quelltextbasis aus.

Um diese Merkmale zu ermöglichen, wurden in der Klassenhierarchie von ObjectWindows einige Änderungen vorgenommen. Wenn Sie Anwendungen über ObjectWindows 1.0, entwickelt haben, hilft Ihnen dieses Kapitel, Ihren existierende Code-Basis auf einfache Weise nach ObjectWindows 2.5 zu konvertieren, so daß Sie von der neuen Funktionalität profitieren können. Zusätzlich haben wir ein Kommandozeilenwerkzeug mit Namen OWLCVT bereitgestellt, das die häufigsten Änderungen, die Sie vornehmen müssen, automatisiert.

Der Begriff ObjectWindows 1.0 wird verwendet, um auf die Version 1.0*x* der Klassenbibliothek von ObjectWindows zu verweisen, die mit dem Paket Borland C++ 3.1 und dem Application Frameworks-Paket bereitgestellt wurde.

Die Zahl von Änderungen, die in Ihrem Code benötigt werden, hängt davon ab, welche Merkmale von ObjectWindows 1.0 Sie in Ihrer speziellen Anwendung benutzen. Es gibt zwar einige Änderungen, die bei *jedem* ObjectWindows 1.0-Programm vorgenommen werden müssen, die meisten Änderungen müssen aber nur stattfinden, wenn Sie ein bestimmtes Merkmal verwendet haben. Mit Hilfe der in diesem Kapitel vorgestellten Checkliste können Sie die betroffenen Bereiche Ihres Codes schnell festlegen und lediglich die Dokumentation für diese Bereiche lesen.

Hinweis für Beta- und EEP-Teilnehmer:

Wenn Sie Anwendungen von ObjectWindows 1.0 nach ObjectWindows 2.5 konvertieren, würden wir gerne von Ihnen hören! Geben Sie bitte Ihre Meinungen und Vorschläge über den Konvertierungsprozeß an Borland via CompuServe in Ihrem geschlossenen Forum weiter. Welche Änderungen an dieser Dokumentation und an dem OWLCVT-Werkzeug könnten den Konvertierungsprozeß erleichtern?

Dieses Kapitel ist in vier Teile organisiert:

- Der Abschnitt *Konvertieren Ihres Codes* erläutert die Verwendung des OWLCVT-Werkzeuges einschließlich der Kommandozeilensyntax, wie OWLCVT Code verändert usw.
- Der Abschnitt *Checkliste zur Konvertierung* beschreibt alle Änderungen, die Sie an Ihrer Anwendung eventuell vornehmen müssen. Zusammen mit dieser Beschreibung liegen Seitenverweise vor, die angeben, wo Sie mehr Information über die erforderliche Änderung finden können. Dadurch müssen Sie nur für die benötigten Änderungen nachlesen und Sie können diejenigen ignorieren, die nicht auf Ihre Anwendung zutreffen.
- Der Abschnitt *Prozeduren zur Konvertierung* enthält detaillierte Beschreibungen von allen Änderungen, die Sie in Ihrer Anwendung eventuell vornehmen müssen. Dies sind die Abschnitte, auf die im ersten Teil verwiesen wird.
- Der Abschnitt *Fehlerquellen* listet eine Reihe allgemeiner Probleme auf, die bei der Konvertierung Ihres Codes von ObjectWindows 1.0 nach ObjectWindows 2.5 auftreten können.

Konvertieren Ihres Codes

Es sind eine Reihe von Schritten erforderlich, um Ihren ObjectWindows 1.0-Code für eine Zusammenarbeit mit der Klassenbibliothek von ObjectWindows 2.5 zu portieren:

- Stellen Sie sicher, daß Ihr Code unter Borland C++ 4.5 geeignet compiliert werden kann. Sie müssen dazu Ihren Code nicht linken und ausführenkönnen, sondern ihn nur ohne Fehler oder Warnungen compilieren.

- Konvertieren Sie Ihren Code mit Hilfe des OWLCVT-Werkzeuges
- Führen Sie die notwendigen manuellen Konvertierungen durch.

Dieser Abschnitt erläutert diese Schritte und dafür benötigten Werkzeuge.

Konvertieren nach Borland C++ 4.5

Bevor Sie Ihren Code konvertieren können, müssen Sie sicherstellen, daß er mit dem Compiler von Borland C++ 4.5 korrekt compiliert werden kann. Änderungen am vorläufigen ANSI C++ Standard, einschließlich dem Hinzufügen von drei verschiedenen **char**-Typen und einer neuen Syntax bei der Verwendung der **new**- und **delete**-Operatoren, um Arrays mit Objekten zu reservieren, könnten die Ursache dafür sein, daß Ihr Code nicht compilierbar ist. Diese Änderungen der Sprache und die Frage, wie die damit assoziierten Probleme angegangen werden können, werden in der Datei README im Abschnitt mit dem Titel *C/C++ Sprachänderungen* beschrieben.

Sie müssen Ihren Code STRICT-verträglich machen. Windows 3.1 führte in WINDOWS.H eine Unterstützung ein, um STRICT zu definieren. Dies aktiviert eine strike Fehlerprüfung beim Compiler. Ein Code, der mit definiertem STRICT geschriben ist, läßt sich leichter zwischen Umgebungen und von 16-Bit- nach 32-Bit-Windows portieren. Weiterinformationen darüber, wie Ihr Code STRICT-verträglich gemacht wird, finden Sie in Kapitel über die Windows-Programmierung im *Programmierhandbuch*.

Sie können Ihre existierenden Make-Dateien, Konfigurationsdateien, Antwortdateien usw. für den Compilierungsprozeß verwenden. *Konfigurationsdateien* sind Dateien, die eine Zahl von Kommandozeilen-Compileroptionen enthalten. *Antwortdateien* sind Dateien, die sowohl Kommandozeilen-Compileroptionen und Dateinamen enthalten. Konfigurationsdateien und Antwortdateien werden eingehender im *Benutzerhandbuch* beschrieben. Die einzigen Änderungen, die Sie für diesen Zweck an Ihren Dateien vornehmen müssen, sind:

- Ändern Sie den Include-Pfad für Header-Dateien. Um Klassen von ObjectWindows 1.0 und zu ObjectWindows 1.0 kompatible Container-Klassen geeignet zu definieren, müssen Sie die folgenden Änderungen vornehmen:

  Ändern Sie C:*BC31*\OWL\INCLUDE in C:*BC4*\INCLUDE\OWLCVT

  Ändern Sie C:*BC31*\CLASSLIB\INCLUDE in C:*BC4*\INCLUDE\CLASSLIB\OBSOLETE

  Ändern Sie C:*BC31*\INCLUDE in C:*BC4*\INCLUDE

Dabei wird davon ausgegangen, daß die existierenden Pfade in Ihren zu ObjectWindows 1.0 kompatiblen Dateien das Verzeichnis C:*BC31* als das Stammverzeichnis Ihrer alten Borland C++ Installation verwenden und

daß Sie Borland C++ 4.5 im Verzeichnis C:*BC4* installiert haben. Die Pfadnamen müssen deshalb gegebenenfalls geändert werden.

- Ihre Include-Pfade sollten in folgender Reihenfolge vorliegen:
  C:*BC4*\INCLUDE\OWLCVT
  C:*BC4*\INCLUDE\CLASSLIB\OBSOLETE
  C:*BC4*\INCLUDE
- Da Sie nur sicherstellen müssen, daß Ihr Code unter Borland C++ 4.5 compilierbar ist, sollten Sie alle Kommandos zum Linken von Ihrer Make-Datei oder Ihrem Skript entfernen:

  Wenn Sie explizite Kommandos zum Linken haben, können Sie diese einfach löschen, auskommentieren oder, falls Sie MAKE verwenden, die passenden .OBJ-Dateien als Ziel in der MAKE-Kommandozeile angeben.

  Wenn Sie den Compiler verwenden, um den Linker automatisch für Sie aufzurufen, fügen Sie die Option **-c** (um das automatische Aufrufen des Linkers zu unterdrücken) Ihren Compiler-Kommandos hinzu.

Wenn Sie irgendwelche Compiler-Fehlermeldungen oder Warnungen erhalten, wenn Sie Ihren Code mit Hilfe von BCC.EXE compilieren, korrigieren Sie die Probleme und compilieren Sie von neuem. Wenn Ihr Code einmal korrekt compiliert, sind Sie bereit für die Konvertierung Ihres Codes nach ObjectWindows 2.5.

Konvertierungen mit OWLCVT

OWLCVT ist ein Kommandozeilenwerkzeug für die Konvertierung Ihres existierenden ObjectWindows 1.0-Codes, um die neuen Klassenbibliotheken von ObjectWindows 2.5 zu nutzen. Es führt eine Reihe von Konvertierungen an Ihrem ObjectWindows 1.0-kompatiblen Quelltext und Header-Dateien durch:

- Es ändert den Mechanismus zur Ereignisbehandlung von DDVT zu Ereignisantworttabellen.
- Es ändert Aufrufe der Hierarchie *TWindowsObject/TWindow* zu Aufrufen der Hierarchie *TWindow/TFrameWindow*.
- Es erhält die Aufrufe der ursprünglichen API-Funktionen von Windows.
- Es bindet die passenden Header-Dateien für ObjectWindows 2.5-Ressourcen ein.
- Es bindet die passenden Header-Dateien für ObjectWindows 2.5-Quelltext ein.
- Es ersetzt Aufrufe von *DefWndProc*, *DefCommandProc*, *DefChildProc* und *DefNotificationProc* durch einen Aufruf der Funktion *DefaultProcessing*.

- Es fügt Kommentare in Ihren Code ein, wenn es auf ein fragliches Konstrukt trifft, das eventuell nicht geändert werden muß.
- Es legt Sicherheitskopien von allen originalen Quelltexten oder Header-Dateien an, die durch OWLCVT verändert wurden.

Kommandozeilensyntax von OWLCVT

Die Kommandozeilensyntax für OWLCVT ist wie folgt:

```
OWLCVT [options] file1 [file2 [file3 [...]]]
```

Dabei ist *filen* eine oder mehrere Quelltextdateien von ObjectWindows 1.0 und *options* eine oder mehrere Kommandozeilen-Compiler-Optionen. OWLCVT akzeptiert alle regulären Kommandozeilen-Compiler-Optionen (BCC.EXE). Damit können Sie alle Ihre alten Kommandoskripten, Make-Dateien, Konfigurationsdateien usw. für die Konvertierung verwenden. Nur einige wenige dieser Optionen besitzen für OWLCVT selbst einen funktionalen Effekt. Aber einige Optionen bewirken, daß Makros in den Header-Dateien von Borland C++ definiert werden. Deshalb sollten Sie dieselben Optionen für die Konvertierung Ihrer Dateien weiter verwenden, die Sie auch schon für deren Compilierung verwendet haben.

Sicherheitskopien für Ihre alten Quelltextdateien

Wenn Sie OWLCVT starten, legt es in Ihrem aktuellen Verzeichnis ein Verzeichnis mit Namen OWLACK an. Anschließend kopiert es Ihre originale Quelltextdatei und alle lokalen Header-Dateien und legt diese im Verzeichnis OWLBACK ab. Wenn OWLCVT mit dem Konvertieren Ihrer Dateien fertig ist, sind die geänderten Quelltextdateien in Ihrem aktuellen Verzeichnis. Wenn die konvertierten Dateien aus irgendeinem Grund nicht korrekt funktionieren, unvollständig sind oder auf andere Weise nicht zufriedenstellen, können Sie Ihre original Dateien auf einfache Weise wieder herstellen, indem Sie sie aus dem Verzeichnis OWLBACK kopieren. Wenn Sie OWLCVT erneut starten und es findet eine Kopie einer Datei, die schon im Verzeichnis OWLBACK vorliegt, bleibt die Kopie im Verzeichnis unberührt und wird nicht überschrieben.

Verwenden von OWLCVT

Wenn Sie Ihren Code mit OWLCVT konvertieren wollen, gibt es eine Reihe durchzuführender Schritte:

1. Kopieren Sie die Datei, die die Compiler-Optionen für Ihre Compilierungen unter ObjectWindows 1.0 enthält (wie z.B. Make-Datei, Konfigurationsdatei, Antwortdatei usw.), in eine neue Datei.
2. Führen Sie die folgenden Änderungen an der neuen Datei durch:

Wenn Sie nicht schon bereits die Include-Pfade der Header-Dateien bei der Konvertierung nach Borland C++ 4.5 geändert haben, dann ändern Sie die Include-Pfade wie folgt:

Ändern Sie C:*BC31*\OWL\INCLUDE in C:*BC4*\INCLUDE\OWLCVT (für ObjectWindows 1.0-kompatible Header-Dateien)

Ändern Sie C:*BC31*\CLASSLIB\INCLUDE in C:*BC4*\INCLUDE\CLASSLIB\OBSOLETE (für Header-Dateien mit *Object*-basierten Containerklassen)

Ändern Sie C:*BC31*\INCLUDE in C:*BC4*\INCLUDE (für Standard-Header-Dateien)

Dabei wird davon ausgegangen, daß die existierenden Pfade in Ihren zu ObjectWindows 1.0 kompatiblen Dateien das Verzeichnis C:*BC31* als das Stammverzeichnis Ihrer alten Borland C++ Installation verwenden und daß Sie Borland C++ 4.5 im Verzeichnis C:*BC4* installiert haben. Ändern Sie die Pfadnamen, um die aktuellen Verzeichnisse zu , in denen Ihr Compiler installiert ist.

Warnung! Wenn Sie irgendwelche Header-Dateien verwenden, die einen gleichen Namen wie den einer Header-Datei von Borland besitzen, *müssen* Sie das Verzeichnis mit diesen Dateien in den Include-Pfad Ihrer Header-Datei *vor* den Include-Verzeichnissen von Borland plazieren. Sie müssen dies tun, selbst wenn die Dateien im aktuellen Verzeichnis liegen und Sie die Syntax **#include** "*filename*.h" für das Einbinden dieser Dateien verwenden.

Wenn Sie eine Make-Datei, eine Batch-Datei oder irgendeinen anderen Typ von Kommandoskript verwenden, dann entfernen Sie alle Kommandos außer denen für die C++ Compilierungen, dem Einbinden beim Linken, Ressource-Compilierung und -Binden usw.. Nehmen wir zum Beispiel an, Sie haben die folgende Batch-Datei:

```
BCC -WS -c -ml -w MYAPP.CPP
RC -r -iC:\BC31\OWL\INCLUDE -iC:\BC31\INCLUDE MYAPP.RC
TLINK /Tw /c /C COWL MYAPP, MYAPP, , @@MAKE0000.$$$, MYAPP.DEF
```

Nach dem Entfernen aller Kommandos außer denen für die C++ Kompilierungen würde diese Datei wie folgt aussehen:

```
BCC -WS -c -ml -w MYAPP.CPP
```

Konvertieren Sie die Compilierungskommandos in OWLCVT-Kommandos. Nehmen wir zum Beispiel an, Sie hätten die Batch-Datei im vorherigen Schritt konvertiert. Nach der Konvertierung des Compilierungskommandos in ein OWLCVT-Kommando würde die Datei wie folgt aussehen:

```
OWLCVT -WS -c -ml -w MYAPP.CPP
```

3. Starten Sie das passende Kommandozeilenwerkzeug; d.h. wenn Sie eine Batch-Datei verwenden, starten Sie die Batch-Datei, wenn Sie eine Make-Datei verwenden, starten Sie MAKE usw.. Wenn Sie eine Konfi-

gurationsdatei oder Antwortdatei von der Kommandozeile aus verwenden, starten Sie OWLCVT genauso wie Sie den Compiler starten würden. Wenn Sie zum Beispiel die Konfigurationsdatei MYCONVRT.CFG hätten und die Datei MYFILE.CPP konvertieren wollten, würde die OWLCVT-Kommandozeile wie folgt aussehen:

```
OWLCVT +MYCONVRT.CFG MYFILE.CPP
```

4. Wenn alle Ihre Dateien von OWLCVT verarbeitet wurden, sollten Sie überprüfen, ob weitere Änderungen erforderlich sind. Diese Änderungen werden im nächsten Abschnitt beschrieben.
5. Wenn Sie alle notwendigen manuellen Änderungen durchgeführt haben, dann erzeugen Sie Ihr Projekt über die Werkzeuge von Borland C++ 4.5.

Checkliste zur Konvertierung

Dieser Abschnitt präsentiert eine Reihe von Konvertierungen, die Sie eventuell nach dem Ablauf von OWLCVT an Ihrem ObjectWindows 1.0-Code durchführen müssen. Die meisten dieser Konvertierungen sind nur notwendig, wenn Sie ein spezielles Merkmal von ObjectWindows 1.0 verwenden. OWLCVT führt eine Zahl von Konvertierungen auch automatisch durch. Die folgenden Konvertierungen müssen manuell durchgeführt werden, allerdings *nur* wenn Sie ein spezielles Merkmal oder Klasse verwenden:

- **Konstruieren virtueller Basen**
  Eine Reihe von Klassen wurden in ObjectWindows für die Verwendung virtueller Basisklassen verändert.
- **Umwandeln virtueller Basen zu abgeleiteten Typen**
  Verwenden Sie das Makro DYNAMIC_CAST, um ein Zeiger auf eine virtuelle Basisklasse zu eine abgeleiteten Klasse umzuwandeln (zum Beispiel eine Übergabe von *TWindow* * an der Stelle von *TFrameWindow* *).
- **Änderungen von Object-basierten Containern zur BIDS-Bibliothek hin**
  Die Klassenbibliothek für *Object*-basierte Container wird in ObjectWindows 2.5 nicht benutzt.
- **Streaming**
  An der Bibliothek für Streams wurden eine Reihe von Änderungen vorgenommen.

- **MDI-Klassen**
  Wenn Sie die Klassen *TMDIFrame* und *TMDIClient* benutzen, müssen Sie eine Reihe von Änderungen vornehmen.
- **Die Variable MainWindow**
  Sie sollten die Variable *TApplication::MainWindow* nicht mehr länger setzen. Stattdessen sollten Sie die Funktion *SetMainWindow* verwenden.
- **Verwenden eines Dialogfensters als Hauptfenster**
  Eine Reihe von Änderungen sind notwendig, wenn Sie ein Dialogfenster als Hauptfenster verwenden.
- **Die Botschaftsverarbeitungsfunktionen von TApplication**
  Die Funktionen *ProcessDlgMsg*, *ProcessAccels* und *ProcessMDIAccels* wurden von der Klasse *TApplication* entfernt.
- **Die Funktion Paint**
  Die Deklaration der Elementfunktion *Paint* von *TWindow* wurde geändert.
- **Die Funktionen CloseWindow, ShutDownWindow und Destroy**
  Die Deklaration dieser Elementfunktionen von *TWindow* wurde geändert.
- **Die Funktionen ForEach und FirstThat**
  Die Deklaration der Elementfunktionen *ForEach* und *FirstThat* von *TWindow* wurde geändert.
- **Die Klassen TComboBoxData und TListBoxData**
  Einige Datenelemente der Klassen *TListBoxData* und *TComboBoxData* wurden geändert.
- **Die Klassen TEditWindow und TFileWindow**
  Die Klassen *TEditWindow* und *TFileWindow* wurden durch *TEditSearch* und *TEditFile* ersetzt.
- **Die Klassen TSearchDialog und TFileDialog**
  Die Klassen *TSearchDialog* und *TFileDialog* wurden durch die Klassen *TReplaceDialog* oder *TFindDialog* und *TFileOpenDialog* ersetzt.
- **Die Funktion ActivationResponse**
  Die Funktion *ActivationResponse* wurde von den Klassen *TWindow* und *TWindowsObject* entfernt. Beispiele dafür, wie dieselbe Funktionalität in ObjectWindows 2.5 erreicht werden kann, finden Sie an der entsprechenden Stelle.
- **Die Funktionen BeforeDispatchHandler und AfterDispatchHandler**
  Die Funktionen *BeforeDispatchHandler* und *AfterDispatchHandler* wurden von ObjectWindows entfernt. Beispiele dafür, wie dieselbe Funktionalität in ObjectWindows 2.5 erreicht werden kann, finden Sie an der entsprechenden Stelle.

- **Die Funktion DispatchAMessage**
  Die Funktion *DispatchAMessage* wurde von ObjectWindows entfernt.
- **Das Datenelement KBHandlerWnd**
  Das Datenelement *KBHandlerWnd* wurde von der Klasse *TApplication* entfernt.
- **MAXPATH**
  MAXPATH ist in keiner Header-Datei von ObjectWindows mehr definiert. Es ist nun allein in der Header-Datei dir.h definiert.
- **Stilkonventionen**
  ObjectWindows 2.5 verwendet etwas andere Stilkonventionen als ObjectWindows 1.0. Auch wenn Ihre Anwendung ohne die stilistischen Änderungen korrekt compilieren müßte, sollten Sie diese Änderungen dennoch vornehmen, um eine einfache Kompatibilität mit Ihrem zukünftigen ObjectWindows 2.5-Code zu gewährleisten.

Prozeduren zur Konvertierung

Dieser Abschnitt enthält detaillierte Beschreibungen der Prozeduren, die in den vorangehenden Abschnitten angerissen wurden. Sie können in diesem Abschnitt wesentlich genauere technische Information über jede Prozedur finden.

Behandlung von Botschaften und Ereignissen

Die in ObjectWindows 1.0 verwendeten DDVTs (*dynamic dispatch virtual tables*) besitzen einige Einschränkungen, speziell bei mehrfacher Vererbung und 32-Bit-Umgebungen. ObjectWindows 2.5 ersetzt DDVTs durch *Ereignisantworttabellen*, die gegenüber DDVTs die folgenden Vorteile bieten:

- Volle Unterstützung der mehrfachen Vererbung bei Fensterklassen
- Automatisches Extrahieren von Botschaften
- Typenüberprüfung aller Funktionen zur Ereignisbehandlung und extrahierter Botschaftsparameter zur Compile-Zeit
- Kompatibilität zwischen 16-Bit- und 32-Bit-Umgebungen
- Leichtere Verwendung von benutzerdefinierten und zur Laufzeit definierten Botschaften
- Möglichkeit, zwei oder mehr Botschaften an eine einzelne Funktion zur Ereignisbehandlung zuzuteilen.
  Volle Verträglichkeit mit dem vorläufigen ANSI C++ Standard

OWLCVT konvertiert Ihre existierenden DDVTs automatisch in Antworttabellen von ObjectWindows 2.5. OWLCVT behält *nicht* Ihre symbolischen Konstanten bei und konvertiert sie stattdessen zu ihren numeri-

schen Werten. Nehmen wir zum Beispiel an, Sie haben die folgende DDVT-Deklaration:

```
virtual void CMTest(TMessage& Msg) = [CM_FIRST + CM_TEST];
```

Wenn OWLCVT dies konvertiert, verwendet es den numerischen Wert des definierten CM_TEST:

```
DEFINE_RESPONSE_TABLE1(TMyWindow, TFrameWindow)
  EV_COMMAND(101, CMTest),
END_RESPONSE_TABLE;
```

Die folgenden Abschnitte beschreiben die manuelle Konvertierung von DDVTs zu Antworttabellen. Es wird allerdings nicht empfohlen, diese Aufgabe manuell durchzuführen, speziell bei großen Anwendungen.

Die folgenden Abschnitte beschreiben nur, wie Ihre existierenden ObjectWindows DDVTs zu konvertieren sind. Antworttabellen bieten mehr Merkmale, die Sie vielleicht nützen wollen. Siehe Kapitel 5 für vollständige Details über Ereignisantworttabellen.

Das Erzeugen einer Ereignisantworttabelle besteht aus vier Schritten, die die folgenden Abschnitte beschreiben:

- Entfernen der DDVT-Funktionen
- Hinzufügen der Deklaration einer Ereignisantworttabelle
- Hinzufügen der Definition einer Ereignisantworttabelle
- Hinzufügen von Einträgen für die Ereignisantworttabelle

Entfernen der DDVT-Funktionen

Sie sollten zuerst die DDVT-Funktionsdeklarationen von Ihrer Fensterklassendefinition entfernen. Sie müssen den DDVT-Zuteilungsindex entfernen (z.B. CM_FIRST + CM_SENDTEXT), da die Definition der Elementfunktion diesen Index nicht benutzt. Der zweite Teil des Zuteilungsindex wird verwendet, wenn Sie Ihre Antworttabelle definieren. Sie können auch das Schlüsselwort **virtual** entfernen, da es bei Ereignisantworttabellen nicht erforderlich ist, daß Funktionen zur Beantwortung von Ereignissen virtuell sind.

Hier sind einige DDVT-Funktionsdeklarationen und deren äquivalente Ereignisantworttabelle:

ObjectWindows 1.0

```
virtual void CMSendText(TMessage &Msg) =
  [CM_FIRST + CM_SENDTEXT];
virtual void CMEmpInput(TMessage &Msg) =
  [CM_FIRST + CM_EMPINPUT];
virtual void HandleListBoxMsg(TMessage &Msg) =
  [ID_FIRST + ID_LISTBOX];
virtual void WMInitMenu(RTMessage) =
  [WM_FIRST + WM_INITMENU];
```

ObjectWindows 2.5

```
void CmSendText();

void CmEmpInput();

void HandleListBoxMsg(UINT);

void EvInitMenu(WPARAM);
```

```
virtual void BNClicked(RTMessage Msg) =          void BNClicked();
  [NF_FIRST + BN_CLICKED];
```

Jede vordefinierte Windows-Botschaft besitzt eine assoziierte spezifische Funktion zur Botschaftsbehandlung. Ferner hat jede Funktion ein spezifisches Aussehen, das Sie berücksichtigen müssen, wenn Sie Ihren eigenen Code für die Behandlung dieser Botschaften schreiben. Die Windows-Botschaften und deren Korrespondierenden Funktionsnamen und Aussehen sine im entsprechenden Kapitel im ObjectWindows *Referenzhandbuch* aufgeführt.

Wenn Sie selbstdefinierte Windows-Botschaften verwenden, können Sie den Funktionsnamen frei wählen. Sie geben den Funktionsnamen mit Hilfe einer der Antworttabellenmakros, die in der entsprechenden Tabelle beschrieben sind, an. Das Aussehen der Funktion hängt von dem verwendeten Makro ab. Siehe das ObjectWindows *Referenzhandbuch* für weitere Information.

Namenskonventionen

Sie sollten die Funktionsnamen von ObjectWindows 2.5 zur Behandlung von Ereignissen mit einem Präfix versehen, das mit einem Großbuchstaben und einem Kleinbuchstaben beginnt und das vom Botschaftstyp (wie WM, EV, CM usw.) übernommen wurde. Zwei Großbuchstaben sollten Sie vermeiden. Zum Beispiel wird aus *CMCommand* dann *CmCommand*. Die vordefinierten Funktionen von ObjectWindows 2.5 zur Behandlung von Ereignissen sind allesamt auf diese Weise benannt.

Hinzufügen der Deklaration einer Ereignisantworttabelle

Der nächste Schritt ist das Hinzufügen der Deklaration einer Ereignisantworttabelle hinter die letzte Deklaration in Ihrer Fensterklasse. Ein Beispiel:

```
class TMyWindow: public TFrameWindow {
  :
  DECLARE_RESPONSE_TABLE(TMyWindow);
  };
```

DECLARE_RESPONSE_TABLE ist ein Makro, das den Klassennamen als Parameter erwartet.

Hinzufügen der Definition einer Ereignisantworttabelle

In Verbindung mit dem Makro DECLARE_RESPONSE_TABLE müssen Sie auch eine Definition einer Ereignisantworttabelle in der Quelltextdatei (*nicht* die Header-Datei) hinzufügen, in der Sie die Elemente Ihrer Fensterklasse definieren. Sie müssen ebenso Einträge für die Ereignisantworttabelle hinzufügen. Dies wird in den folgenden Abschnitten erläutert. Hier ist eine Beispieldefinition einer Ereignisantworttabelle:

```
// BEACHTE: Antworttabellen sollten in globalem Gültigkeitsbereich
definiert werden.
DEFINE_RESPONSE_TABLE1(TMyWindow, TFrameWindow)
  // Einträge der Ereignisantworttabelle
  .
  .
  .
END_RESPONSE_TABLE;
```

DEFINE_RESPONSE_TABLE*X* ist ein Makro, das den Namen der Fensterklasse und deren direkte Basisklasse als Parameter erwartet. Das *X* basiert auf der Zahl von Basisklassen, die Ihre Klasse besitzt. END_RESPONSE_TABLE ist ein Makro, das die Definition der Ereignisantworttabelle beendet. Siehe Kapitel 5 über mehr Details zu Definitionen von Ereignisantworttabellen.

Hinzufügen von Einträgen für die Ereignisantworttabelle

In ObjectWindows 1.0 bestimmte der Zuteilungsindex, den Sie in der Deklaration einer Elementfunktion zur Beantwortung von Botschaften verwendet haben, auf welche Art von Botschaft die Funktion antwortet. So identifizierte zum Beispiel die Konstante CM_FIRST Elementfunktionen zur Beantwortung von Kommandos.

Ereignisantworttabellen von ObjectWindows 2.5 bieten alle Zuteilungstypen von ObjectWindows 1.0 und einiges mehr. Die unten stehende Tabelle listet die Zuteilungstypen von ObjectWindows 1.0 und deren Äquivalente in der Ereignisantworttabelle von ObjectWindows 2.5 auf. Siehe die folgenden Abschnitte für einzelne Informationen zu jedem Zuteilungstyp.

Tabelle A.1: Elementfunktionen für die Beantwortung von Botschaften und Einträge der Ereignisantworttabelle

Elementfunktionstyp für die Beantwortung von Botschaften	Version 1.0 Zuteilungs-konstante	Version 2.5 Eintrag der Ereignisantworttabelle
Kommandobotschaft	CM_FIRST	EV_COMMAND
Auf die untergeordnete Fenster-ID basierende Botschaft	ID_FIRST	EV_CHILD_NOTIFY_ALL_CODES
Benachrichtigungs-basierende Botschaft	NF_FIRST	EV_NOTIFY_AT_CHILD
Windows-Botschaften	WM_FIRST	EV_MESSAGE und EV_WM_*XXX*

Antworten auf Kommandobotschaften

Kommandobotschaften sind solche, für die Windows eine Botschaft WM_COMMAND von einem Menü oder einem Tastenkürzel aus sendet. In ObjectWindows 1.0 sollten Sie eine Elementfunktion deklarieren, die die Summe aus CM_FIRST und der Ressource-ID des Menüs oder Tastenkürzels benutzt; ObjectWindows fängt die Botschaft WM_COMMAND ab und ruft die Elementfunktion zur Beantwortung von Botschaften mit der passenden ID auf.

In ObjectWindows 2.5 passiert dasselbe, wobei allerdings Ereignisantworttabellen anstatt DDVTs verwendet werden. Hier ist ein Beispiel:

```
// Deklaration der ObjectWindows 1.0 Elementfunktion
virtual void CMSendText(TMessage &Msg) =
  [CM_FIRST + CM_SENDTEXT];

// Eintrag der 2.5 Ereignisantworttabelle
EV_COMMAND(CM_SENDTEXT, CmSendText)
void CmSendText();
```

Antworten auf Botschaften mit untergeordneter Fenster-ID

Elementfunktionen zur Beantwortung von Botschaften, die auf der ID untergeordneter Fenster basieren, behandeln alle Botschaften, die von einem Dialogelement kommen und die von ObjectWindows an das zum Dialogelement übergeordnete Fenster weitergereicht wurden. In ObjectWindows 1.0 wurde der Benachrichtigungs-Code des Dialogelementes an das Element *TMessage.LP.Hi* übergeben, das von der Elementfunktion zur Beantwortung von Botschaften überprüft werden mußte (überlicherweise mit einer **switch**-Anweisung).

ObjectWindows 2.5 unterstützt dieselbe Art der Zuteilung durch den Eintrag EV_CHILD_NOTIFY_ALL_CODES in der Ereignisantworttabelle; alle Benachrichtigungs-Codes werden an eine einzelne Elementfunktion übergeben. Hier ist ein Beispiel:

```
// Deklaration der ObjectWindows 1.0 Elementfunktion
virtual void HandleListBoxMsg(TMessage &Msg) =
  [ID_FIRST + ID_LISTBOX]

// Eintrag der ObjectWindows 2.5 Ereignisantworttabelle und
Funktionsdefinition
EV_CHILD_NOTIFY_ALL_CODES(ID_LISTBOX, HandleListBoxMsg)
void HandleListBoxMsg(UINT);
```

ObjectWindows 2.5 unterstützt ebenso die Zuteilung von speziellen Benachrichtigungs-Codes an spezielle Elementfunktionen. Dies ist in ObjectWindows 1.0 nicht möglich. Verwenden Sie für solch eine Zuteilung den Eintrag EV_CHILD_NOTIFY. Hier ist ein Beispiel:

```
EV_CHILD_NOTIFY(ID_BUTTON, HandleButtonClick, BN_CLICKED)
void HandleButtonClick();
```

Da Sie oft auf Benachrichtigungs-Codes von Windows-Dialogelementen antworten müssen, definiert ObjectWindows Makros für eine einfachere Behandlung von Benachrichtigungs-Codes für Schalter, Kombinationsfelder, Editierfelder und Listenfelder. Hier ist ein Beispiel, das den Benachrichtigungs-Code LBN_DBLCLK vereinfacht:

```
EV_LBN_DBLCLK(ID_LISTBOX, HandleListBoxMsg)
void HandleListBoxMsg(UINT);
```

Botschaften, die auf der ID untergeordneter Fenster basieren, sind letztlich Kommandobotschaften, die einen Benachrichtigungs-Code ein-

schließen. Für Kommandoschalter ist der Benachrichtigungs-Code Null, wodurch er einer Menükommandobotschaft ähnelt. Der empfohlene Weg zur Antwort auf Schalterdruck geht über Antwortfunktionen für Kommandobotschaften und nicht über Antwortfunktionen für Botschaften, die auf der ID untergeordneter Fenster basieren. Ein »OK«-Schalter ist beispielsweise ein zu einem Dialogfenster untergeordnetes Fenster. Wenn der Benutzer den Schalter anklickt, übergibt der Schalter eine Botschaft, die wie eine Kommandobotschaft behandelt werden kann. Sie können die Schalterbotschaft wie folgt behandeln:

```
EV_COMMAND(IDOK, CmOk)
```

Antworten auf Benachrichtigungsbotschaften

Benachrichtigungsbotschaften sind mit Botschaften, die auf der ID untergeordneter Fenster basieren, vergleichbar. Sie werden allerdings nicht vom übergeordneten Fenster behandelt, sondern vom Dialogelement selbst. Benachrichtigungsbotschaften sind die beste Wahl für das Erzeugen von spezialisierten Dialogelementklassen.

Sowohl ObjectWindows 1.0 als auch 2.5 teilen die Benachrichtigungsbotschaften speziellen Elementfunktionen zu. Hier ist ein Beispiel:

```
// Deklaration der ObjectWindows 1.0 Elementfunktion
virtual void HandleListBoxMsg(TMessage &Msg) =
  [ID_FIRST + ID_LISTBOX]

virtual void ENChange(TMessage &Msg) = [NF_FIRST + EN_CHANGE]

// Eintrag der ObjectWindows 2.5 Ereignisantworttabelle und
Funktionsdefinition
EV_NOTIFY_AT_CHILD(EN_CHANGE, ENChange)
void FNameChange();
```

Antworten auf generelle Botschaften

Sie können auf Botschaften antworten, die keine Kommandobotschaften, Botschaften, die auf der ID untergeordneter Fenster basieren, oder Benachrichtigungsbotschaften sind.

ObjectWindows 1.0 und 2.5 teilen Windows-Botschaften speziellen Elementfunktionen zu. Beachten Sie, daß die Namenskonvention von ObjectWindows 2.5 für Windows-Botschaften die Verwendung des Präfixes *Ev* vorsieht, wobei eine gemischte Groß/Kleinschreibung der Windows-Botschaftskonstante verwendet wird:

```
// Deklaration der ObjectWindows 1.0 Elementfunktion
virtual void WMCtlColor(TMessage &Msg) = [WM_FIRST + WM_CTLCOLOR]

// Eintrag der ObjectWindows 2.5 Ereignisantworttabelle
EV_MESSAGE(WM_CTLCOLOR, EvCtlColor)
```

Wie bei Botschaften, die auf der ID untergeordneter Fenster basieren, definiert ObjectWindows Makros für eine einfache Antwort auf Windows-Botschaften. Hier ist ein Beispiel, das das vordefinierte Makro für die Botschaft WM_CTLCOLOR benutzt:

```
// Eintrag der ObjectWindows 2.5 Ereignisantworttabelle
EV_WM_CTLCOLOR
```

Siehe das Kapitel für weitere Details über die vordefinierten Makros für Botschaften.

Die Verwendung der vordefinierten Makros geht davon aus, daß Sie Ihre Funktion zur Beantwortung von Ereignissen nach der *Ev*-Namenskonvention benennen. Ein weiterer guter Grund für die Verwendung der vordefinierten Makros liegt darin, daß ObjectWindows die Parameter automatisch »extrahiert«, die normalerweise in den Parametern *LPARAM* und *WPARAM* übergeben werden. So wird beispielsweise bei der Verwendung von EV_WM_CTLCOLOR davon ausgegangen, daß Sie eine Elementfunktion zur Beantwortung von Ereignissen wie folgt deklariert haben:

```
HBRUSH EvCtlColor(HDC hDCChild, HWND hWndChild, UINT nCtrlType);
```

Das Extrahieren der Botschaften stellt eine strikte C++ Typenüberprüfung zur Compile-Zeit bereit. Dadurch können Fehler schon während dem Compilieren aufgedeckt werden und nicht erst zur Laufzeit.

Beispiel einer Ereignisantworttabelle

Hier sind einige Deklarationen von ObjectWindows 1.0 Fensterklassen (auf der linken Seite) und deren Äquivalente in ObjectWindows 2.5 (auf der rechten Seite):

```
class TMyWindow: public TWindow {
  :
  virtual void WMCtlColor(TMessage &Msg) =
     [WM_FIRST + WM_CTLCOLOR];
  virtual void WMPaint(TMessage &Msg) =
     [WM_FIRST + WM_PAINT];

  virtual void CMSendText(TMessage &Msg) =
     [CM_FIRST + CM_SENDTEXT];
  virtual void CMEmpInput(TMessage &Msg) =
     [CM_FIRST + CM_EMPINPUT];
  };


class TMyWindow: public TFrameWindow {
  :
  LPARAM EvMyMessage(WPARAM, LPARAM);
  void EvPaint();

  void CmSendText();
  void CmEmpInput();
```

```
  DECLARE_RESPONSE_TABLE(TMyWindow);
  };

DEFINE_RESPONSE_TABLE(TMyWindow, TFrameWindow)
  EV_MESSAGE(WM_MYMESSAGE, EvMyMessage),
  EV_WM_PAINT,
  EV_COMMAND(CM_SENDTEXT,CmSendText),
  EV_COMMAND(CM_EMPINPUT, CmEmpInput),
END_RESPONSE_TABLE;


class TMyDialog: public TDialog {
  .
  .
  .
  virtual void HandleListBoxMsg(TMessage &Msg) =
     [ID_FIRST + ID_LISTBOX];
  };


class TMyDialog: public TDialog {
  .
  .
  .
  void HandleListBoxMsg(UINT);

  DECLARE_RESPONSE_TABLE(TMyDialog);
  };

DEFINE_RESPONSE_TABLE (TMyDialog, TDialog)
  EV_CHILD_NOTIFY_ALL_CODES(ID_LISTBOX, HandleListBoxMsg),
END_RESPONSE_TABLE;


class TMyButton: public TButton {
  .
  .
  .
  virtual void BNClicked(TMessage &Msg) =
     [NF_FIRST + BN_CLICKED];
  };


class TMyButton: public TButton {
  .
  .
  .
  void BNClicked ();

  DECLARE_RESPONSE_TABLE(TMyButton);
  };

DEFINE_RESPONSE_TABLE(TMyButton, TButton)
  EV_NOTIFY_AT_CHILD(BN_CLICKED, BNClicked),
END_RESPONSE_TABLE;
```

Ändern Ihrer Fensterobjekte

ObjectWindows 1.0 hatte zwei Klassen für »generische« Fenster: *TWindowsObject* und *TWindow*. *TWindowsObject* war eine abstrakte Klasse.

Sie stellte das grundlegende Verhalten für alle Fenster, Dialogfenster und andere Oberflächenelemente bereit. Eine Instanz der Klasse *TWindowsObject* selbst war allerdings nicht sehr nützlich. *TWindow* diente auf der anderen Seite als die Klasse, die Sie für alle Fenstertypen verwendeten. Unglücklicherweise bedeutete das, daß selbst einfache untergeordnete *TWindow*-Objekte Funktionalität und Code besaßen, die sie nicht nutzten.

ObjectWindows 2.5 bietet zwei neue Klassen: *TWindow* und *TFrameWindow*. *TWindow* ähnelt der Klasse *TWindowsObject* in ObjectWindows 1.0, sie ist allerdings nicht abstrakt. Sie können Instanzen von *TWindow* in ObjectWindows 2.5 für untergeordnete Fenster verwenden. *TFrameWindow*-Objekte als überlappende oder Popup-Hauptfenster. Sie verwalten ein Client-Fenster und sind durch *TMDIFrame* für MDI-Unterstützung sowie von *TDecoratedFrame* für die Unterstützung von *Dekorationen* (wie Mauspaletten und Statuszeilen) beerbt.

Konvertieren von Konstruktoren

OWLCVT führt eine Suchen-und-Ersetzen-Operation auf Ihren Quelltextdateien durch, wodurch alle Vorkommnisse von *TWindow* durch *TFrameWindow* und alle Vorkommnisse von *TWindowsObject* durch *TWindow* ersetzt werden. Diese Modifikation ist allerdings nicht ausreichend. Der Konstruktor von *TFrameWindow* erwartet nicht immer dieselben Parameter wie der alte Konstruktor von *TWindow*. Für die Klasse *TWindow* in ObjectWindows 1.0 gab es zwei Konstruktoren:

```
TWindow(PTWindowsObject, LPSTR, PTModule = NULL);
TWindow(HWND, PTModule = NULL);
```

OWLCVT konvertiert den Namen *TWindow* zu *TFrameWindow*. Allerdings korrespondiert nach dieser Konvertierung keiner dieser Konstruktoren direkt mit den in *TFrameWindow* verfügbaren Konstruktoren:

```
TFrameWindow(TWindow *parent,
             const char far *title = 0,
             TWindow *clientWnd = 0,
             BOOL shrinkToClient = FALSE,
             TModule *module = 0);
TFrameWindow(HWND hWnd, TModule *module = 0);
```

Die allgemeinsten Verwendungen des Konstruktors von *TWindow* in ObjectWindows 1.0 waren wie folgt:

```
// Erster Konstruktor von TWindow, der Parameter PTModule auf seinen
Vorgabewert setzen.
TWindow(AParent, "Title");

// Zweiter Konstruktor von TWindow, der Parameter PTModule auf seinen
Vorgabewert setzen.
TWindow(AParent);
```

OWLCVT konvertiert diese Aufrufe nach :

```
TFrameWindow(parent, "Title");
TFrameWindow(parent);
```

Diese Aufrufe werden korrekt compiliert. Der erste Aufruf setzt die letzten drei Parameter des fünfstelligen Konstruktors von *TFrameWindow* auf deren entsprechenden Vorgabewerte. Der zweite Aufruf setzt den zweiten Parameter des zweistelligen Konstruktors von *TFrameWindow* auf seinen Vorgabewert. Sie müssen keine weiteren Änderungen vornehmen, es sei denn Sie legen fest, daß Sie eine Wertangabe für einen der anderen Parameter benötigen.

Wenn Ihr ObjectWindows 1.0 Code einen Wert für den Parameter *PTModule* festlegt, könnte die Konvertierung Ihres Konstruktors, wie sie von OWLCVT vorgenommen wurde, nicht mit einem gültigen Konstruktor von *TFrameWindow* korrespondieren. Der Konstruktor von *TWindow* könnte zum Beispiel in etwa wie folgt aussehen:

```
TWindow(AParent, "Title", ptModule);
TWindow(AParent, ptModule);
```

Der konvertierte Code würde wie folgt aussehen:

```
TFrameWindow(parent, "Title", ptModule);
TFrameWindow(parent, ptModule);
```

Der zweite Aufruf compiliert und funktioniert korrekt. Damit der erste Aufruf korrekt arbeitt, können Sie zum einen einfach die Variable *ptModule* vollständig entfernen:

```
TFrameWindow(parent, "Title");
```

Auf diese Art erhalten die restlichen drei Parameter des fünfstelligen Konstruktors ihre Vorgabewerte. Sie können auch Vorgabewerte für den dritten und vierten Parameter angeben:

```
TFrameWindow(parent, "Title", 0, FALSE, ptModule);
```

Siehe im ObjectWindows *Referenzhandbuch* unter der Klasse *TFrameWindow* für weitere Informationen zu den Konstruktoren von *TFrameWindow* und deren Parameter.

Aufrufen von Windows-API-Funktionen

ObjectWindows 2.5 kapselt viel mehr von der Windows-API als ObjectWindows 1.0. Der Vorteil besteht darin, daß ObjectWindows für die Übergabe allgemeiner Parameter an die API-Funktionen sorgt wie beispielsweise bei Fenster-Handles. Da aber einige Funktionen in ObjectWindows 2.5 dieselben Namen wie Windows-API-Funktionen besitzen, könnten Sie Compile-Zeit-Fehler wie den folgenden bekommen:

```
Extra parameter in call to TClass::MessageBox(const char far *, const
char far
*, unsigned int)
```

Der einfachste Weg, Ihren Code fehlerfrei zu bekommen, ist die Verwendung des Zugriffsoperators **::**. Nehmen wir zum Beispiel an, Sie haben in Ihrer ObjectWindows 1.0-Anwendung die Windows-API-Funktion *MessageBox* auf folgende Weise aufgerufen:

```
void TMyWindow::CMAddRecord() {
  MessageBox(HWindow, "All fields must be filled in", "Input Error", MB_OK);
  }
```

Sie können diese Funktion durch Hinzufügen der Zugriffsoperators **::** dazu veranlassen, die Windows-API-Funktion mit ObjectWindows 2.5 aufzurufen:

```
void TMyWindow::CMAddRecord() {
  ::MessageBox(HWindow, "All fields must be filled in", "Input Error",
MB_OK);
  }
```

Sie können ebenso die gekapselte API-Funktion *TWindow::MessageBox* verwenden:

```
void TMyWindow::CMAddRecord() {
  MessageBox("All fields must be filled in", "Input Error", MB_OK);
  }
```

Der Vorteil des gekapselten ObjectWindows-Äquivalentes besteht darin, daß Sie keine Fensterparameter explizit übergeben müssen. Diese werden von den Elementfunktionen von *TWindow* behandelt, die von der Klasse geerbt wurden, welche Sie für den Aufruf verwenden. OWLCVT stellt allen Aufrufen von Windows-API-Funktionen als Präfix den Zugriffsoperator **::** voran.

Ändern von Header-Dateien

Sie müssen zwei Änderungen an der Art und Weise vornehmen, wie Sie Header-Dateien in Ihrem Code einbinden:

- Verwenden der neuen Positionen für Header-Dateien
- Verwenden der neuen gestrafften Header-Dateien von ObjectWindows

Verwenden der neuen Positionen für Header-Dateien

Borland C++ 4.5 legt alle Header-Dateien im Verzeichnis INCLUDE ab. Header-Dateien von ObjectWindows liegen jetzt im Verzeichnis INCLUDE\OWL. Die Header-Dateien für die Containerklassenbibliothek und Laufzeitbibliothek liegen ebenso im Verzeichnis INCLUDE.

Bei Versionen von Borland C++ vor 4.5 müßten Sie Ihren Include-Verzeichnispfad in etwa wie folgt gesetzt haben:

```
C:\BORLANDC\INCLUDE; C:\BORLANDC\OWL\INCLUDE; C:\BORLANDC\CLASSLIB\INCLUDE
```

Bei Version 4.5 benötigen Sie nur noch:

```
C:\BORLANDC\INCLUDE
```

Anstatt Header-Dateien in Ihren Code mit Befehlen wie

```
#include <applicat.h>
```

einzubinden, binden Sie nun die Header-Dateien der ObjectWindows Header-Dateien wie folgt ein:

```
#include <owl\applicat.h>
```

oder

```
#include <classlib\arrays.h>
```

Alle Quelltexte und Beispielanwendungen der ObjectWindows verwenden diesen Ansatz.

Sie können auch Ressourcen-Script-Dateien und Ressoucen-Header-Dateien auf diese Art einbinden. Um zum Beispiel die Ressoucen-Header- und Ressourcen-Script-Dateien für *TPrinter* einzubinden, würde die **#include**-Anweisung wie folgt aussehen:

```
#include <owl\printer.rh>
#include <owl\printer.rc>
```

Verwenden der neuen gestrafften Header-Dateien von ObjectWindows

Header-Dateien in ObjectWindows 2.5 enthalten weniger Klassen als deren Entsprechungen in ObjectWindows 1.0. Da weniger Klassen in jeder Datei deklariert sind, müssen Sie möglicherweise mehr Header-Dateien explizit einbinden. So bewirkte zum Beispiel das Einbinden von owl.h in ObjectWindows 1.0 die Definition mehrerer Klassen. einschließlich *TWindowsObject*, *TWindow*, *TMDIFrame*, *TMDIClient* und *TDialog*. Die Funktionalität, die durch das Einbinden von owl.h entsteht, kann erreicht werden, indem man applicat.h, framewin.h, dialog.h, mdi.h, scroller.h und dc.h einbindet.

Die Header-Datei owlall.h bindet alle Header-Dateien von ObjectWindows ein, die für das Erzeugen einer vorcompilierten Header-Datei von ObjectWindows nützlich sein können. Zum Beispiel erzeugt das folgende Fragment eine vorcompilierte Header-Datei mit Hilfe von owlall.h:

```
#pragma hdrfile "OWLALL.CSM"
#include <owl\owlall.h>
#pragma hdrstop
```

Der Vorteil der Verwendung vorcompilierter Header-Dateien besteht darin, daß sie einen großen Anstieg der Compiliergeschwindigkeit ermöglichen, da sie die Zeit für die Verarbeitung der Header-Dateien um bis zu 90% reduzieren. Für weitere Informationen zu vorcompilierten Header siehe das Kapitel im *Benutzerhandbuch*.

ObjectWindows-Ressourcen

ObjectWindows 1.0 kombinierte die Ressourcen und Bezeichner, die von mehreren Klassen benutzt wurden, in nur wenigen Dateien. Wenn Ihre

Anwendung die Ressourcen einer Klasse benutzte, standen Ihnen die Ressourcen auch für eine Reihe anderer Klassen zur Verfügung, ob Sie sie benutzten oder nicht. In ObjectWindows 2.5 liegt eine Ressourcenskriptdatei und eine Ressourcen-Header-Datei (Eine Ressourcen-Header-Datei enthält alle diejenigen Bezeichner für die Ressourcen, die in der Ressourcenskriptdatei definiert sind) für jede Klasse vor, die Ressourcen erfordert.

Dadurch wird das unnötige Einbinden von Ressourcen oder Header-Dateien verhindert. Die Namen der Ressourcenskriptdateien und Ressourcen-Header-Dateien gleichen den korrespondierenden Header-Dateinamen. Die Klasse *TPrinter* ist zum Beispiel in der Header-Datei printer.h definiert. Die Ressourcen-IDs für die Klasse *TPrinter* sind in der Datei printer.rh enthalten. Die von der Klasse *TPrinter* benutzten Ressourcen sind in der Datei PRINTER.RC enthalten.

Compilieren von Ressourcen

Wenn Sie Ihre Ressourcen compilieren, sollten Sie mit Änderungen des Include-Pfades für Header-Dateien für den Ressourcen-Compiler vorsichtig sein. Der Include-Pfad für Header-Dateien in ObjectWindows 1.0 bindet für gewöhnlich die Verzeichnisse C:*BC31*\INCLUDE C:*BC31*\OWL\INCLUDE und C:*BC31*\CLASSLIB\INCLUDE mit ein. Für Borland C++ 4.5 sollte dieser Pfad geändert werden, damit C:*BC4*\INCLUDE und OWL\, die dem Dateinamen vorangehen, durchsucht wird.

Dabei wird davon ausgegangen, daß die existierenden Pfade in Ihren zu ObjectWindows 1.0 kompatiblen Dateien das Verzeichnis C:*BC31* als Stammverzeichnis Ihrer alten Installation von Borland C++ verwenden und daß Sie Borland C++ 4.5 im Verzeichnis C:*BC4* installiert haben. Sie müssen diese Pfade gegebenenfalls entsprechend ändern.

Um die Ressourcen für eine ObjectWindows-Klasse mit einzubringen, können Sie einfach die passende Ressourcendatei von Ihrer eigenen Ressourcen-Script-Datei einbinden. Um zum Beispiel Ressourcen für die Klasse *TPrinter* hinzuuzufügen, würden Sie die folgende Zeile in Ihrer .RC-Datei hinzufügen:

```
#include <owl\printer.rc>
```

Menü-Ressourcen

Wenn Sie Menü-Ressourcen in Ihrem Code verwenden, könnte eine Änderung dessen erforderlich sein, wie Menüs Ihren Rahmenfensterobjekten zugewiesen werden. ObejctWindows 1.0 erlaubt Ihnen die direkte Zuweisung eines Menü an ein Rahmenfensterobjekt, indem das Element *Menu* der Struktur *Attr* des Objektes mit einer speziellen Ressource-ID gleichgesetzt wurde. Zum Beispiel:

```
Attr.Menu = MENU_1;
```

ObjectWindows 2.5 läßt diese Form der Zuweisung nicht mehr zu. Stattdessen sollten Sie die Funktion *TFrameWindow::AssignMenu* verwenden. Die oben stehende Zeile sieht dann mit der Funktion *AssignMenu* wei folgt aus:

```
AssignMenu(MENU_1);
```

Konstruieren virtueller Basen

Eine Reihe von Klassen, die in ObjectWindows 1.0 nicht-virtuelle Basisklassen erwarteten, sind in ObjectWindows 2.5 von virtuellen Basisklassen abgeleitet. Zum Zwecke der Portierung verwenden die davon betroffenen Klassen *TWindow* und *TFrameWindow* als virtuelle Basisklassen. Diese Klassen sind: *TDialog*, *TMDIFrame*, *TFrameWindow*, *TMDIChild*, *TDecoratedFrame*, *TLayoutWindow*, *TClipboardViewer*, *TKeyboardModeTracker* und *TTinyCaption*. In C++ werden virtuelle Basisklassen zuerst konstruiert. Das bedeutet, daß der Konstruktor der abgeleiteten Klasse keine Standardargumente für den Basisklassenkonstruktor angeben kann. Siehe Mehtoden für den Umgang beim Konstruieren virtueller Basen.

Umwandeln virtueller Basen zu abgeleiteten Typen

Eine allgemein gebräuchliche Praktik bei ObjectWindows 1.0-Code ist die Umwandlung eines Zeigers auf *TWindowsObject* in einen abgeleiteten Typ. Die Basisklasse *TWindow* (das Äquivalent zu *TWindowsObject* in ObjectWindows 2.5) ist bei vielen Standardklassen von ObjectWindows 2.5 eine virtuelle Basis; Die Sprache C++ erlaubt Ihnen allerdings nicht, einen Zeiger auf eine virtuelle Basisklasse zu einer abgeleiteten Klasse umzuwandeln. Um diesen Konstrukttyp nach ObjectWindows 2.5 zu konvertieren, müssen Sie das Makro DYNAMIC_CAST benutzen. Das Makro DYNAMIC_CAST erwartet zwei Parameter. Der erste Parameter ist der Datentyp, zu dem Sie hin umwandeln wollen. Der zweite Parameter ist die Klasseninstanz, die Sie auf eine untergeordnete Klasse umwandeln wollen.

Zum Beispiel wandelt der folgende Code den Zeiger auf ein *TWindowsObject*-Objekt *Parent* in ein *TWindow* um:

```
TMyChildWindow::MyFunc() {
  .
  .
  .
  // Parent ist aktuell ein TWindowsObject-Objekt.
  ((TWindow *)Parent)->AssignMenu("NewMenu");
  .
  .
  .
  }
```

Sie könnten dies konvertieren, indem Sie einfach die Klasse *TWindow* in eine Klasse *TFrameWindow* konvertieren:

```
TMyChildWindow::MyFunc() { // Fehler bei nächster Zeile
  .
  .
  .
```

```
// Parent ist aktuell ein TWindow-Objekt.
((TFrameWindow *)Parent)->AssignMenu("NewMenu");
.
.
.
}
```

In ObjectWindows 2.5 ist der Typ von *Parent* ein *TWindow* * (es war ein *TWindowsObject* *), welches eine virtuelle Basis von *TFrameWindow* ist. Ein Versuch, dies auf eine untergeordnete Klasse umzuwandeln, endet mit einem Compile-Zeit-Fehler. Der korrekte Weg, dies zu konvertieren geht über das Makro DYNAMIC_CAST, zum Beispiel wie folgt:

```
TMyChildWindow::MyFunc() {
.
.
.
DYNAMIC_CAST(Parent,TFrameWindow)->AssignMenu("NewMenu");
.
.
.
}
```

Hier ist die Syntax für das Makro DYNAMIC_CAST:

```
type DYNAMIC_CAST(type, object)
```

Dabei gilt:

- *type* ist der Datentyp, zu dem hin Sie das Objekt umwandeln wollen.
- object ist das Objekt, das Sie umwandeln wollen.

Wenn die Konvertierung erfolgreich ist, gibt DYNAMIC_CAST *object* als ein *type*-Datenobjekt zurück. Wenn die Konvertierung fehlschlägt, ist das Ergebnis des Markos DYNAMIC_CAST der Wert Null. Sie sollten eine Fehlerüberprüfung durchführen, wenn Sie das Makro DYNAMIC_CAST verwenden.

Überwechseln von Object-basierten Containern zur BIDS-Bibliothek

Bei ObjectWindows 1.0 war die Klasse *TWindowsObject* von der Klasse *Object* aus der Containerklassenbibliothek abgeleitet. Bei ObjectWindows 2.5 wird die mit Templates versehene BIDS Containerklassenbibliothek ans Stelle der *Object*-basierten Containerklassenbibliothek verwendet. Die BIDS-Bibliothek bietet schnellere Ausführungszeiten und wesentlich größere Flexibilität für den Code. Die mit Templates versehenen BIDS-Containerklassen sind im Kapitel des *Programmierhandbuches* beschrieben. Diese Änderung betrifft Code, der die Klasse *TWindow* (das Äquivalent von *TWindowsObject* bei ObjectWindows 2.5) in *Object*-basierte Container plaziert, sowie Code, der Elementfunktionen von *Object* wie beispielsweise *IsA*, *NameOf* und die Elementfunktionen *isXXX* wie *isEmpty*, *isFull*, *isSortable* usw. aufruft.

Code, der die Klasse *TWindow* in *Object*-basierte Container plaziert, sollte konvertiert werden, um die mit Templates versehenen BIDS-Containerklassen zu verwenden. Anderenfalls müßten Sie, um Ihre eigenen *TWin-*

dow-Klassen in *Object*-basierte Container abzulegen, folgendes durchführen:

- Leiten Sie Ihre Klasse mehrfach von *Object* wie auch von seiner ObjectWindows-Basisklasse ab.
- Implementieren Sie die Umwandelbarkeit für Ihre Klasse; siehe die Datei README für Informationen zu dieser Prozedur.
- Wenn Sie die Umwandelbarkeit implementieren (was wärmstens empfohlen ist), dann verwenden Sie das Makro DYNAMIC_CAST, um die *Object*-Elemente aus dem Container zurück zu Ihrer von *TWindow* abgeleiteten Klasse umzuwandeln.

Streaming

An der Stream-Klassenbibliothek wurden einige geringfügige Änderungen vorgenommen. Ebenso wurden an der Implementierung des Streaming für ObjectWindows 2.5-Klassen substantielle Änderungen vorgenommen, auch wenn existierender Code mit nur geringen Modifikationen weiterhin korrekt arbeiten sollte.

Entfernte Inserter- und Extractor-Operatoren>>)

Folgende Operatoren existieren nicht mehr:

```
opstream &operator <<(opstream &, TStreamable *);
ipstream &operator >>(ipstream &, void * &);
```

Wenn Sie diesen **<<**-Operator aufgerufen hatten, können Sie jetzt stattdessen den folgenden Aufruf verwenden:

```
opstream.WriteObjectPtr((TStreamable *) p);
```

Dieser **>>**-Operator wurde entfernt, da er keine reale Funktionalität besaß.

Implementieren von Streaming

Die Containerklassenbibliothek von Borland C++ 4.5 vereinfacht das Einstellen Ihrer Klasse auf Streaming erheblich. Der Prozeß verwendet dazu die Makros DECLARE_STREAMABLE und IMPLEMENT_STREAMABLEX.

Das Makro DECLARE_STREAMABLE kann in einer Klasse eingesetzt werden, die von *TStreamable* abgeleitet ist (wie es die meisten Klassen von ObjectWindows sind). Es erwartet zwei Parameter: den Klassennamen und die Versionsnummer. Ein Beispiel:

```
class TMyClass : public TStreamable {
  DECLARE_STREAMABLE(TMyClass, 1);
  };
```

Was Sie als Versionsnummer verwenden, hängt ganz von Ihnen ab. Einige Funktionen zum Streaming geben die Versionsnummer währen bestimmter Operationen aus. Sie *müssen* das Makro DECLARE_STREAMA-

BLE in Ihre Klassendefinition einfügen, damit Sie die Funktionalität für das Streaming bei Ihren ObjectWindows-Klassen benutzen können.

Nachdem Sie Ihre Klasse mit dem Makro DECLARE_STREAMABLE als streamfähig deklariert haben, müssen Sie noch das Makro IMPLEMENT_STREAMABLEX angeben. Dieses Makro führt eine Reihe von Schritten aus, die das Streaming Ihrer Klasse ermöglichen. Dazu zählt auch das Erzeugen eines Extractor-Operators für Ihre Klasse:

```
ipstream & operator >>(ipstream &, TMyClass * &);
```

Beim Makro IMPLEMENT_STREAMABLE*X* müssen Sie *X* angeben, um festzulegen, welches Makro Sie verwenden wollen. Zählen Sie die Zahl der unmittelbaren Basisklassen für Ihre Klasse plus die Zahl der virtuellen Basisklassen, für die Sie das Streaming haben wollen. Diese Zahl gibt an, welches Makro Sie verwenden. Nehmen wir zum Beispiel an, die Klasse *TMyClass* ist von *TFrameWindow* abgeleitet, welche von *TWindow* virtuell erbt. In diesem Falle würden Sie das Makro IMPLEMENT_STREAMABLE2 verwenden.

Sie müssen auch Funktionen zum Lesen und Schreiben für Ihre Klasse bereitstellen. Ein Beispiel:

```
void MyClass::Write(opstream &) {
  // Was auch immer für eine Funktionalität Sie benötigen
  }

void * MyClass::Read(ipstream &, unsigned long ) {
  // Was auch immer für eine Funktionalität Sie benötigen
  .
  .
  .
  }
```

Für weitere Information über die Makros DECLARE_STREAMABLE und IMPLEMENT_STREAMABLE*X* und über Streaming-Klassen im allgemeinen siehe das Kapitel im *Programmierhandbuch*.

MDI-Klassen

TWindow enthielt in ObjectWindows 1.0 alle notwendigen Hilfen, um sie als untergeordnetes MDI-Fenster einsetzen zu können. Dadurch wird die Erzeugung einer MDI-Anwendung vereinfacht. Allerdings wird dadurch auch der MDI-spezifische Code eingebunden, selbst wenn er von Ihrer Anwendung nicht benötigt wird. ObjectWindows 2.5 stellt drei verschiedene MDI-Klassen bereit: *TMDIFrame*, *TMDIClient* und *TMDIChild*. Damit kann Ihre Anwendung *nur dann* MDI-spezifischen Code einbinden, wenn sie auch MDI-Klassen verwendet.

In ObjectWindows 1.0 arbeitete eine typische MDI-Anwendung wie folgt:

- Eine Instanz einer spezialisierten *TMDIFrame*-Klasse diente als Hauptfenster der Anwendung.

- Instanzen spezialisierter *TWindow*-Klassen, die in das Rahmenfenster eingefügt wurden, dienten als untergeordnete MDI-Fenster.

In ObjectWindows 2.5 wird auf ähnliche Weise vorgegangen:

- Eine Instanz der Klasse *TMDIFrame* dient als Hauptfenster der Anwendung.
- Eine Instanz von *TMDIClient* dient als das MDI-Client-Fenster.
- Instanzen der Klasse *TMDIChild*, die in das Client-Fenster eingefügt werden, dienen als untergeordnete MDI-Fenster.

Es gibt eine Reihe von Beispielen mit Namen MFILEAPP und MDITEST, die die MDI-Merkmale verwenden. Diese Beispiele liegen in den Verzeichnissen EXAMPLES\OWL\MFILEAPP und EXAMPLES\OWL\MDITEST Ihrer Installation von Borland C++.

Bilden des Rahmens und des Client

In ObjectWindows 1.0 bestand eine typische Verwendung von *TMDIFrame* darin, eine Klasse von *TMDIFrame* abzuleiten und eine Instanz dieser Klasse in *TApplication::InitMainWindow* zu erzeugen. In ObjectWindows 2.5 können Sie einfach ein *TMDIFrame*-Stammobjekt als Hauptfenster festlegen. Der Standardparameter TMDIClient& für den Konstruktor von *TMDIFrame* erzeugt ein *TMDIClient*-Standardobjekt. Wenn Sie einen spezialisierten Typen von *TMDIClient* benötigen, können Sie ein *TMDIClient*-Objekt erzeugen und es an den Konstruktor selbst übergeben. Die Verwendung einer von *MDIFrame* abgeleiteten Klasse begünstigt die Portierung Ihres Codes. Aber Ihre neuen Anwendungen für ObjectWindows 2.5 dürften keinen spezialisierten *TMDIFrame*-Objekte benötigen.

Der folgende Code zeigt, wie MDI-Clients und untergeordnete Fenster in ObjectWindows 1.0 typischerweise behandelt werden:

```
class TMyMDIFrame : public TMDIFrame {
public:
  TMyMDIFrame(LPSTR title,LPSTR menuName);
  };

void TMyApp::InitMainWindow() {
  SetMainWindow(new TMyMDIFrame("Main Window", "MENU_1"));
  }
```

In ObjectWindows 2.5 würde dieser Code wie folgt aussehen:

```
void TMyApp::InitMainWindow() {
  SetMainWindow(new TMDIFrame("Main Window", "MENU_1"));
  }
```

Wenn Sie ein selbstdefiniertes MDI-Client-Fenster angeben wollen, müßten Sie den Code nur geringfügig ändern:

```
class TMyMDIClient : public TMDIClient {
public:
  TMyMDIClient();
  };

void TMyApp::InitMainWindow() {
  SetMainWindow(new TMDIFrame("Main Window", "MENU_1",
                              *new TMyMDIClient));
  }
```

Der Grund dafür, warum der Konstruktor von *TMDIFrame* eine Referenz auf ein *TMDIClient* erwartet und nicht einen Zeiger, liegt darin, daß dadurch verhindert wird, daß Sie ein *TMDIFrame* mit einem Nullzeiger auf ein *MDIClient* konstruieren. Die Verwendung des Referenzparameters bietet größere Sicherheit, da dann von Ihnen die Bereitstellung eines aktuellen Objektes verlangt wird.

Bilden eines untergeordneten Fensters

In ObjectWindows 1.0 wurde in untergeordnetes Fenster typischerweise wie folgt erzeugt:

```
void TMyMDIFrame::MakeNewChild() {
  PTWindow * newMDIChild = new TMyChild(this, "new child");
  GetApplication()->MakeWindow(newMDIChild);
  }
```

In ObjectWindows 2.5 sollte diese Funktion ein Element der auf *TMDIClient* basierenden Klasse sein:

```
void TMyMDIClient::MakeNewChild() {
  (new TMyMDIChild(*this, "new child"))->Create();
  }
```

Sie müssen für untergeordnete MDI-Fenster *TMDIChild* oder eine von *TMDIChild* abgeleitete Klasse verwenden. Beachten Sie, daß ***this*** als erster Parameter an den Konstruktor von *TMDIChild* übergeben wird. Es sei nochmals erwähnt, daß untergeordnete MDI-Fenster ein *TMDIClient* als übergeordnetes Fenster haben. Damit erwarten ihre Konstruktoren eine Referenz auf *TMDIClient* und keinen Zeiger.

WB_MDICHILD

Das Flag WB_MDICHILD ist nicht mehr definiert. Damit wurde angegeben, ob eine *TWindow*-Klasse wirklich ein untergeordnetes MDI-Fenster war, bzw. bei *TMDIFrame*, welche untergeordneten Fenster wirklich untergeordnete MDI-Fenster und welche nicht waren (ein Mauspaletten zum Beispiel würde nicht als ein untergeordnetes MDI-Fenster implementiert). In ObjectWindows 2.5 gibt es eine *TMDIChild*-Klasse und deren übergeordnetes Fenster ist immer ein *TMDIClient*. Da alle untergeordneten MDI-Fenster von *TMDIChild* abgeleitet und zu *TMDIClient* untergeordnete Fenster sind und da Mauspaletten u.ä. ein zu *TDecoratedMDIFrame* untergeordnetes Fenster darstellen, wird dieses Flag nicht mehr benötigt.

Umgelegte Funktionen

Die folgenden Funktionen der Klasse *TMDIFrame* für den Umgang mit untergeordneten Fenstern wurden zur Klasse *TMDIClient* umgelegt:

```
ArrangeIcons                CMCreateChild
CascadeChildren             CMInitChild
CloseChildren               CMTileChildren
CMArrangeIcons              CreateChild
CMCascadeChildren           InitChild
CMCloseChildren             TileChildren
```

Der Code, bei dem auf diese Funktionen zugegriffen wurde bzw. bei dem sie überschrieben wurden, sollte geändert werden, um die Instanz von *TMDIClient* zu referenzieren, oder auf einen Nachkommen der Klasse *TMDIClient* umgelegt werden.

Die Namen der Funktionen zur Behandlung von Menükommandos sind nach dem ObjectWindows 2.5-Stil gebildet, d.h. *CMInitChild* ist jetzt *CmInitChild*.

Ersetzen von ActiveChild durch GetActiveChild

Bei ObjectWindows 1.0 konnten Sie das aktive untergeordnete MDI-Fenster über das *PTWindow*-Datenelement *ActiveChild* des *TMDIFrame*-Objektes finden. In ObjectWindows 2.5 sollten Sie die Elementfunktion *GetActiveChild* in der Klasse *TMDIClient* verwenden.

Die Variable *MainWindow*

Sie sollten nicht mehr weiter die Variable *TApplication::MainWindow* setzen. Stattdessen sollten Sie die Funktion *SetMainWindow* verwendne. *SetMainWindow* erwartet ein *TFrameWindow* * als Parameter und gibt einen Zeiger auf das alte Hauptfenster zurück. Wenn es sich um eine neue Anwendung handelt (d.h. eine Anwendung, die noch kein Haupfenster eingerichtet hat), ist der Rückgabewert Null.

Angenommen, Ihre existierender Code sieht in etwa wie folgt aus:

```
void InitMainWindow()
{
  MainWindow = new TFrameWindow(0, "This window", new TWindow);
  MainWindow->AssignMenu("COMMANDS");
}
```

In ObjectWindows 1.0 war es allgemein üblich, Ihr Hauptfenster zu Beginn der Ausführung Ihrer Anwendung einzurichten. In ObjectWindows 2.5 sind die Datenelemente der Klasse entweder geschützt oder privat, wodurch Sie die Werte der Datenelemente nicht mehr direkt setzen können. Der oben stehende Code müßte dann in etwa wie folgt aussehen:

```
void InitMainWindow()
{
  SetMainWindow(new TFrameWindow(0, "This window", new TWindow));
  MainWindow->AssignMenu("COMMANDS");
}
```

Verwenden eines Dialogfensters als Hauptfenster

Da die Funktion *SetMainWindow* ein *TFrameWindow* * als Parameter erwartet, ist es nicht mehr möglich, ein *TDialog* bzw. ein von *TDialog* abgeleitetes Objekt als das Hauptfenster zu übergeben. Um das *TDialog*-Objekt als das Hauptfenster zu verwenden, müssen die ein Dialogfenster zu einem Client in einem *TFrameWindow* machen. Dann übergeben Sie dieses *TFrameWindow* als den Parameter an *SetMainWindow*. Das Beispiel CALC im Verzeichnis EXAMPLES\OWL\CALC Ihrer Borland C++ Installation verdeutlicht die Verwendung einer von *TDialog* abgeleiteten Klasse als ein Client-Fenster in einem *TFrameWindow*-Objekt.

Nehmen wie zum Beispeil an, Sie hätten eine von *TDialog* abgeleitete Klasse mit Namen *TMyDialog* konstruier und Sie wollten sie als das Hauptfenster benutzen. Der Code könnte dann in etwa wie folgt aussehen:

```
SetMainWindow(new TFrameWindow(0, "My MainWindow", new TMyDialog, TRUE));
```

Wenn Sie ein Dialogfenster als Ihre Hauptfenster verwenden, sind eine Reihe von Änderungen erforderlich:

- Wenn Sie Ihr Dialogfensterobjekt entfernen, wird der Rahmen nicht mit entfernt. Dies müssen Sie explizit durchführen.
- Sie können nicht mehr Ressourcen direkt dem Dialogfenster hinzufügen, da es nicht das Hauptfenster ist. Se müssen die Ressourcen dem Rahmenfenster hinzufügen. Nehmen wir zum Beispiel an, Sie hatten ein Symbol Ihrem Dialogfenster über die Funktion *SetIcon* hinzugefügt. Jetzt müssen Sie die Funktion *SetIcon* für Ihr Rahmenfenster verwenden.
- Sie können in der Ressource selbst nicht mehr den Titel Ihres Dialogfensters festlegen. Stattdessen müssen Sie den Titel über das Rahmenfenster setzen.
- Sie müssen die Eigenschaften des Dialogfensters wie folgt setzen:
  sichtbar (WS_VISIBLE)
  untergeordnetes Fenster (WS_CHILD)
  ohne Schalter für Symbol- und Vollbilddarstellung, Zugleisten, Systemmenüs oder irgendwelche anderen Standardattribute der Rahmenfenster

Die Botschaftsverarbeitungsfunktionen von *TApplication*

Die Funktionen *ProcessDlgMsg*, *ProcessAccels* und *ProcessMDIAccels* wurde in der Klasse *TApplication* entfernt. Die Verarbeitung von Botschaften erfolgt nun über den Aufruf der virtuellen Funktion *ProcessAppMsg* von *TApplication*. Diese Funktion ruft die virtuelle Funktion *TWindow::PreProcessMsg* des Fensters auf, das die Botschaft empfängt, (und weiter die Kette übergeordneter Fenster hinauf) bis irgendeines die Botschaft vorverarbeitet oder bis es keine übergeordneten Fenster mehr

gibt. An dieser Stelle überprüft sie die Tastenkürzeltabelle der Anwendung und teilt die Botschaft schließlich dem Fenster zu, sofern die Botschaft nicht schon bearbeitet wurde. Diese Änderung vereinfacht und automatisiert die Prozedur der Botschaftsverarbeitung enorm.

Sie könnten Code für ObjectWindows 1.0 besitzen, in dem *ProcessAppMsg* überschrieben wurde, damit die Ordnung der Aufrufe anderer Verarbeitungsfunktionen geändert wird. Ein solcher Fall liegt bei dem Beispiel CALC aus ObjectWindows 1.0 vor. Dieser Code ist in ObjectWindows 2.5 wahrscheinlich überflüssig. Wenn Sie ihn trotzdem in einem speziellen Fall benötigen, sollte dazu die Funktion *PreProcessMsg* des *TWindow*-Objektes oder die Funktion eines seiner übergeordneten Fenster überschrieben werden.

Sie könnten auch Code für ObjectWindows 1.0 besitzen, der *ProcessAccels* erweitert, um mit mehreren Tastenkürzeltabellen für verschiedene Fenster zu arbeiten. Dies wird am besten dadurch geändert, daß Sie jedem Fenster eine Tastenkürzeltabelle zuweisen. Dadurch verarbeitet das Fenster die Tabelle automatisch. Sie können ebenso die Funktion *PreProcessMsg* für die *TWindow*-Klasse bzw. für jede von *TWindow* abgeleitete Klasse überschreiben, damit jede ihre eigene Tastenkürzeltabelle bearbeitet.

Die Funktion GetModule

Die Funktion *GetModule* wurde von der Klasse *TWindowsObject* entfernt. In den meisten Fällen können Sie einen Aufruf von *GetModule* durch einen von *GetApplication* ersetzen. Nehmen wir zum Beispiel an, Sie haben den folgenden Code in Ihrer ObjectWindows 1.0-Anwendgungapplication:

```
GetModule()->ExecDialog(new TDialog(this, "DIALOG_1"));
```

Sie können dies nach ObjectWindows 2.5 konvertieren, indem Sie *GetModule* in *GetApplication* ändern:

```
GetApplication()->ExecDialog(new TDialog(this, "DIALOG_1"));
```

Obwohl ObjectWindows 2.5 aus Kompatibilitätsgründen *ExecDialog* zur Verfügung stellt, wäre die dafür empfohlene Methode eine direkte Verwendung des Kommandos *Execute* aus der instanziierten Klasse. Damit würde sich der oben stehende Code wie folgt ändern:

```
TDialog(this, "DIALOG_1")).Execute();
```

Auf diese Änderung wird genauer eingegangen.

Eine Ausnahme dazu liegt vor, wenn für den *TWindow*-Nachfolger kein *TApplication*-Objekt bzw. kein von *TApplication* abgeleitetes Objekt definiert ist (wie bei einer DLL, die von einer ObjectWindows-Anwendung nicht benutzt wird) und wenn Sie eine Elementfunktion von *TModule* benötigen. Verwenden Sie in diesem Falle das Modulobjekt, das Sie für die DLL in Ihrer Funktion *LibMain* konstruieren. Ein Beispiel:

```
// Deklariere einen globalen Zeiger auf TModule.
TModule *dllModule;

int FAR PASCAL LibMain( ⋮ ) {
  ⋮
  // Weise dllModule hier einen Wert zu.
  dllModule = new TModule("My module", instance, cmdLine);
  ⋮
  }

void MyFunc() {
  TWindow *parentAlias;

  // Verwende die Funktion GetParentObject mit dllModule.
  parentAlias = dllModule->GetParentObject(HWnd);
  }
```

Siehe DLLHELLO.CPP im Verzeichnis EXAMPLES\OWL\MISC Ihrer Installation von Borland C++ für ein detailliertes Beispiel.

Die Funktionen DefXXXProc

Die Elementfunktionen *DefCommandProc*, *DefChildProc*, *DefNotificationProc* und *DefWndProc* von *TWindowsObject* wurden von *TWindow* entfernt (das Äquivalent von *TWindowsObject* in ObjectWindows 2.5) und durch die einzige Funktion *DefaultProcessing* effektiv ersetzt. Dies vereinfacht die Botschaftsverarbeitung erheblich. Um die Standardverarbeitung auszulösen, müssen Sie nur Ihre Basisklassenversion der von Ihnen überschriebenen Ereignisbehandlung oder *DefaultProcessing* aufrufen.

Überschreiben

Im allgemeinen sollten Sie am besten genau ein Kommando bzw. Benachrichtigung, basierend auf der ID eines untergeordneten Fensters, je Funktion bearbeiten. Manchmal kann es aber nützlich sein, mehrere Botschaften mit einer Funktion zu behandeln. Wenn Sie *DefCommandProc* oder *DefChildProc* für diese Zwecke überschrieben hatten, gibt es zwei Möglichkeiten, diesen Code zu portieren:

- Überschreiben Sie die Funktion *EvCommand* und führen die Botschaftsbehandlung dort durch. Das Beispiel CALC im Verzeichnis EXAMPLES\OWL\CALC Ihrer Installation von Borland C++ verdeutlicht, wie dies geschieht. Dies ist keine reine Standardverarbeitung, da der Aufruf von *EvCommand* vor der Suche nach einer Ereignisbehandlung stattfindet.
- Überschreiben Sie die Funktion *DefWindowProc* und fangen Sie die Kommandos dort ab. *DefWindowProc* wird aufgerufen, wenn keine Ereignisbehandlung gefunden wurde.
- Code, der *DefWndProc* überschreibt, überschreibt auch *DefWindowProc*. Code, der *DefNotificationProc* überschreibt, muß mit Hilfe des Makros

EV_NOTIFY_AT_CHILD portiert werden, damit jede Benachrichtigung beim untergeordneten Fenster mit einer separaten Elementfunktion behandelt wird.

Verwenden von DefWndProc für registrierte Botschaften

Wenn Sie *DefWndProc* überschrieben hatten, um registrierte Windows-Botschaften zu behandeln (Botschaften, die durch *RegisterWindowMessage* zurückgegeben wurden), müssen Sie dies in ObjectWindows 2.5 nicht durchführen. Siehe die Beschreibung des Makros EV_REGISTERED.

Die Funktion Paint

Die Deklaration der Elementfunktion *Paint* von *TWindow* hat sich geändert von:

```
virtual void Paint(HDC, PAINTSTRUCT _FAR &);
```

zu:

```
virtual void Paint(TDC&, BOOL, TRect &);
```

TDC ist ein Bestandteil der GDI-Kapselung der Windows-API in ObjectWindows 2.5. Sie können den TDC-Parameter in gleicher Weise wie den HDC-Parameter verwenden. Für die Klasse *TDC* ist ein **operator** *HDC()* definiert, der ein *TDC* in ein *HDC* konvertiert. Der BOOL- und TRect&-Wert korrespondiert direkt mit den Elementen *fErase* und *rcPaint* des PAINTSTRUCT-Typs. Die Datenelemente werden in der Funktion *TWindow::EvPaint* initialisiert. Diese Funktion wird durch die Standardverarbeitungsfunktionen aufgerufen, wenn eine WM_PAINT-Botschaft empfangen wird. Die Funktion *EvPaint* ruft wiederum die Funktion *Paint* auf.

Nehmen wir zum Beispiel an, Ihre Code für ObjectWindows 1.0 enthielt die folgende Funktionsdeklaration:

```
void TWindow::Paint(HDC hdc, PAINTSTRUCT& ps) {
  // Hier viel Code
⋮
  }
```

Sie würden dies für ObjectWindows 2.5 wie folgt ändern:

```
void Paint(TDC& tdc, BOOL erase, TRect& rect) {
  // Hier viel Code
⋮
  }
```

Die Funktionen CloseWindow, ShutDownWindow und Destroy

Die Deklarationen für diese Elementfunktionen von *TWindow* haben sich geändert. Die Versionen dieser Funktionen, die keinen Parameter erwarteten, wurden dahingehend modifiziert, daß sie nun ein **int** erwarten. Diese Funktionen stellen allerdings auch den Vorgabewert für den **int**-Pa-

rameter bereit. Deshalb müßte Ihr existierender Code ohne Änderungen compilierbar und ausführbar bleiben.

Die Funktionen ForEach und FirstThat

Die Funktionen *ForEach* und *FirstThat* werden benutzt, um über die zu einem Fensterobjekt untergeordneten Fenster zu iterieren. Sie müssen beim Aufruf einen Zeiger auf eine Iterator-Funktion als ersten Parameter der Funktion *ForEach* und *FirstThat* übergeben. Diese Iterator-Funktion kann eine normale Funktion oder die Elementfunktion einer Klasse sein. In ObjectWindows 1.0 übergaben die Funktionen *ForEach* und *FirstThat* an die Iterator-Funktionen ein **void *** für deren ersten Parameter. Die Iterator-Funktionen mußten dann diesen **void *** in ein *TWindow* * umwandeln. Dies funktioniert, wenn der korrekte Parametertyp übergeben wird. Eine Typenüberprüfung wird allerdings dabei nicht durchgeführt. In ObjectWindows 2.5 erwarten diese Funktionen direkt ein *TWindow* * :

Beispiele von Iterator-Funktionen für die Funktion *ForEach*:

ObjectWindows 1.0

```
void MyIterator(void *, void *)
void TMyClass::MyIterator(void *, void *)
```

ObjectWindows 2.5

```
void MyIterator(TWindow *, void *)
void TMyClass::MyIterator(TWindow *, void *)
```

Beispiele von Iterator-Funktionen für die Funktion *FirstThat*:

ObjectWindows 1.0

```
BOOL MyIterator(void *, void *)
BOOL TMyClass::MyIterator(void *, void *)
```

ObjectWindows 2.5

```
BOOL MyIterator(TWindow *, void *)
BOOL TMyClass::MyIterator(TWindow *, void *)
```

Die Funktionen werden immer noch auf gleiche Weise verwendet:

```
void TMyWindow::SomeMyFunc() {
  ForEach( MyIterator , 0 );
  ForEach( &TMyClass::MyIterator , 0 );
  }
```

Die Klassen TComboBoxData und TListBoxData

In ObjectWindows 1.0 besaß die Klasse *TListBoxData*, die eine Transferstruktur für *TListBox* darstellt, folgende zwei Datenelemente:

```
PArray Strings;
PArray SelStrings;
```

Diese Elemente waren Zeiger auf *Array*-Objetke, die auf *Object* basierten, und hielten Instanzen der auf *Object* basierenden Klasse *String*. Diese Instanzen würden die Strings im Listenfeld und die ausgewählten Strings (meistens bei Listenfeldern mit mehrfacher Auswahl verwendet) sein. Mit der Vorgabe der Änderung einer auf *Object* basierenden Klassenbibliothek zu BIDS-Bibliotheken, auf Templates basieren, und der Einführung einer Klasse *string* durch das ANSI-Kommittee wurde die Implementierung dieser Datenelemente für ObjectWindows 2.5 wie folgt geändert:

```
TStringArray *Strings;
TStringArray *SelStrings;
```

TStringArray verwendet eine BIDS-Array-Klasse, um ein Array mit *string*-Objekten zu halten.

Eine ähnliche Änderung existiert bei *TComboBoxData*: das Datenelement *Strings* ist ein Zeiger auf *TStringArray* an Stelle eines Zeigers auf *Array*.

Obwohl die neue ANSI-Klasse *string* viele neue Operatoren und Funktionen bietet, fehlt ein **const char *** -Operator, der in einer auf *Object* basierenden Klasse vorkommt. Stattdessen besitzt sie eine Elementfunktion *c_str*, die verwendet werden muß, um die Daten der Klasse zu erhalten. Damit sind Änderungen bei dem Code notwendig, der auf dem **const char *** -Operator der auf *Object* basierenden *String*-Klasse beruht. Ebenso müssen Sie dort ein *TStringArray* benutzen, wo Sie eine auf *Object* basierende *Array*-Klasse verwendet hatten, um Daten aus der Struktur *TListBoxData* zu erhalten.

Nehmen wir zum Beispiel an, Sie haben unter ObjectWindows 1.0 gerade einen Transfer vollzogen und erhalten einen **const char *** auf den ersten ausgewählten String. Angenommen *DialogTransfer* ist ein Zeiger auf einen Transferpuffer und *ListBoxData* ein Zeiger auf ein *TListBoxData*-Objekt innerhalb des Puffers.

```
Array& selStrings = *(DialogTransfer->ListBoxData->SelStrings);

const char *sel = (const char *)(String &)selStrings[0];
```

Für ObjectWindows 2.5 wird dies zu:

```
TStringArray& selStrings = *(DialogTransfer->ListBoxData->SelStrings);

const char *sel = selStrings[0].c_str();
```

Die Klassen TEditWindow und TFileWindow

Die Klassen *TEditWindow* und *TFileWindow* von ObjectWindows 1.0 wurden entfernt und bezüglich ihrer Funktionalität durch *TEditSearch* und *TEditFile*, die von der Dialogelementklasse *TEdit* abgeleitet sind, ersetzt. Die Klassen *TEditSearch* und *TEditFile* sind keine vollwertigen Rahmen-

fenster mit Menüs wie bei den vorherigen Klassen. Sie werden aber stattdessen dazu verwendet, *TFrameWindow*- oder *TMDIChild*-Fenstern eine Editor-Funktionalität hinzuzufügen.

Es gibt zwei Methoden, die Sie für das Ersetzen der Instanzen von *TFileWindow* oder *TEditWindow* in Ihrem Code benutzen können: Verwenden der Klassen *TFileWindow* und *TEditWindow*, die im Beispielprogramm OLDFILEW definiert sind oder Hinzufügen der Klassen *TEditSearch* und *TEditFile* als Client-Fenster in *TFrameWindow*- oder *TMDIChild*-Fenstern.

Verwenden des Beispiels OLDFILEW

Die Klassen *TEditWindow* und *TFileWindow* wurden in den Beispielprogrammen EDITWND und FILEWND implementiert. Sie können diese Beispiele im Verzeichnis EXAMPLES\OWL\OLDFILEW Ihrer Installation von Borland C++ finden. Die in diesen Beispielen definierten Klassen *TEditWindow* und *TFileWindow* können fast auf gleiche Weise benutzt werden wie die Originalklassen *TEditWindow* und *TFileWindow* von ObjectWindows 1.0. Um Ihre Programm mit diesen Klassen zu erweitern, müssen Sie den Quelltext in Ihr Quelltextverzeichnis für Ihre Anwendung kopieren. Wenn Sie nur die Klasse *TEditWindow* verwenden, müssen Sie auch nur die Dateien EDITWND.CPP und editwnd.h kopieren. Da die Klasse *TFileWindow* auf der Klasse *TEditWindow* basiert, benötigen Sie für die Klasse *TFileWindow* auch die Dateien FILEWND.CPP und filewnd.h. Ihre Quelltextdateien, die auf diese Klassen verweisen, müssen die passenden Header-Dateien einbinden.

Zwar funktioniert diese Methode für die Konvertierung Ihres Codes. Es wird allerdings empfohlen, daß Sie neuen Code schreiben, der auf die Methode von ObjectWindows 2.5 zurückgreift, die Client-Fenster *TEditSearch* und *TEditFile* in den *TFrameWindow*- oder *TMDIChild*-Fenstern zu verwenden

Hinzufügen von TEditSearch- und TEditFile-Client-Fenstern

Sie können die Funktionalität der Klassen *TEditWindow* und *TFileWindow* erhalten, indem Sie ein *TFrameWindow* oder *TMDIChild* instanziieren und ein *TEditFile*- oder *TEditSearch*-Objekt als Client-Fenster angeben. Beide Klassen, *TFrameWindow* und *TMDIChild*, besitzen einen Konstruktor, der einen Zeiger auf *TWindow* als seinen dritten Parameter erwartet. Er benutzt dann das *TWindow*-Objekt bzw. das von *TWindow* abgeleitete Objekt als ein Client-Fenster. Um ein *TEditFile*- oder *TEditSearch*-Objekt als ein Client für einer dieser Klassen anzugeben, müssen Sie nur das *TEditFile*- oder *TEditSearch*-Objekt konstruieren und einen Zeiger auf das Objekt an den Konstruktor übergeben.

Die folgenden Code-Zeilen stammen aus dem Beispiel FILEAPP im Verzeichnis EXAMPLES\OWL\FILEAPP Ihrer Installation von Borland C++. Sie verdeutlichen, wie ein *TEditFile*-Client-Fenster in einem *TFrameWindow*-Fenster geöffnet wird.

```
void TFileApp::InitMainWindow() {
  :
  SetMainWindow(new TFrameWindow(0, Name, new TEditFile));
  :
  }
```

Die folgenden Code-Zeilen stammen aus dem Beispiel MFILEAPP im Verzeichnis EXAMPLES\OWL\MFILEAPP Ihrer Installation von Borland C++. Sie verdeutlichen, wie ein *TEditFile*-Client-Fenster in einem *TFrameWindow*-Fenster geöffnet wird.

```
void TMDIFileApp::CmFileNew() {
  :
  TMDIChild *child(*Client, "", new TEditFile(0, 0, 0));
  :
  }
```

Die Klassen TSearchDialog und TFileDialog

Die Klassen *TSearchDialog* und *TFileDialog* wurden für ObjectWindows entfernt. Sie sollten die Klasse *TReplaceDialog* oder *TFindDialog* an Stelle von *TSearchDialog* sowie die Klasse *TFileOpenDialog* an Stelle der Klasse *TFileDialog* verwenden. Diese neuen Klassen basieren auf der Klasse *TCommonDialog*, die die grundlegende Funktionalität der allgemeinen Dialogfenster von Windows einkapselt.

Die Funktion ActivationResponse

Die Funktion *ActivationResponse* wurde von den Klassen *TWindow* und *TWindowsObject* entfernt. Um festzustellen, ob ein Fenster aktiviert wurde, können Sie die passende Botschaft wie WM_MDIACTIVATE, WM_ACTIVATE oder WM_SETFOCUS auffangen. Im Beispiel SCRNSAVE können Sie die Verwendung von WM_ACTIVATE, um festzustellen, wann ein Fenster aktiv ist, studieren. Dieses Beispiel findet sich im Verzeichnis EXAMPLES\OWL\SCRNSAVE Ihrer Installation von Borland C++. Eine Verwendung von WM_SETFOCUS können Sie im Beispiel BSCRLAPP im Verzeichnis EXAMPLES\OWL\BSCRLAPP Ihrer Installation von Borland C++ studieren.

Die Funktionen BeforeDispatchHandler und AfterDispatchHandler

Die Funktionen *BeforeDispatchHandle* und *AfterDispatchHandler* wurden in ObjectWindows entfernt. Sie können eine ähnliche Funktionalität erhalten, indem Sie *WindowProc* für eine von *TWindow* abgeleitete Klasse überschreiben. Dazu müssen Sie folgendes durchführen:

1. Überladen Sie die Funktion *WindowProc* in Ihrer abgeleiteten Klasse.
2. Führen Sie Bearbeitungen in Ihrer Funktion *WindowProc* durch, bevor Sie die vorgegebene Funktion *TBaseClass::WindowProc* aufrufen.
3. Rufen Sie *TBaseClass::WindowProc* auf.

4. Sichern Sie den Rückgabewert von *TBaseClass::WindowProc*.
5. Führen Sie Bearbeitungen nach der Ausführung von *TBaseClass::WindowProc* durch.
6. Geben Sie den gesicherten Rückgabewert beim Verlassen Ihrer *WindowProc* zurück.

Ein Beispiel:

```
LRESULT TMyWindow::WindowProc(UINT msg, WPARAM wParam, LPARAM lParam) {
  // hier beliebige Bearbeitung 'vorher'.

  BeforeHandling();

  LRESULT ret = TFrameWindow::WindowProc(message, wParam, lParam);

  // hier beliebige Bearbeitung 'nachher'.
  AfterHandling();

  return ret;
  }
```

Die Funktion DispatchAMessage

DispatchAMessage wurde von ObjectWindows entfernt. Botschaften sollten an die Windows-API mit der Kapselung *SendMessage* von ObjectWindows 2.5 gesendet werden.

Generelle Botschaften

Für das Senden genereller Fensterbotschaften (das sind Botschaften, die nicht Teil von WM_COMMAND sind, wie etwa Botschaften WM_FIRST + *XXX*), würde der Code wie folgt konvertiert werden:

```
// Vorher
DispatchAMessage(WM_MESSAGE, AIMessage, &TWindow::DefWndProc)
// Nachher
SendMessage(WM_MESSAGE, AIMessage.WParam, AIMessage.LParam);

// Vorher
SomeOtherWindow->DispatchAMessage(WM_FIRST + WM_MESSAGE, AIMessage,
                                  &TWindow::DefWndProc)
// Nachher
SomeOtherWindow->SendMessage(WM_MESSAGE, AIMessage.WParam,
                             AIMessage.LParam);
```

Umgang mit dem Parameter DefProc

DispatchAMessage erwartete einen Zeiger auf eine Funktion als ihren letzten Parameter. *DispatchAMessage* rief diese Funktion auf, wenn ein DDVT-Eintrag für die Botschaft nicht gefunden wurde. Wenn ein Fenster von ObjectWindows 2.5 eine Botschaft empfängt und keine Behandlung für sie findet, startet es automatisch eine geeignete Standardbehandlung. Siehe Siehe Standardbehandlungen von Botschaften.

Kommandobotschaften

Es gibt eine Reihe verschiedener Arten von Kommandobotschaften, die Sie eventuell konvertieren müssen. Kommandobotschaften für Menü in der Form CM_FIRST + XXX werden wie folgt konvertiert:

```
// Vorher
OtherWin->DispatchAMessage(CM_FIRST + CM_MENUID,AIMessage,
                           &TWindow::DefCommandProc);
// Nachher
OtherWin->SendMessage(WM_COMMAND, CM_MENUID, AIMessage.LParam);
```

Bei ObjectWindows 2.5 gehen die auf diese Weise gesendeten Kommandobotschaften direkt zum angegebenen Fenster, *nicht* zum Fokus-Fenster.

Benachrichtigungen, die auf der ID untergeordneter Fenster basieren, mit der Form ID_FIRST + XXX werden wie folgt konvertiert:

```
// Vorher
OtherWin->DispatchAMessage(ID_FIRST + ID_CHILDID,AIMessage,
                           &TWindow::DefChildProc);
// Nachher
OtherWin->SendMessage(WM_COMMAND, ID_CHILID, AIMessage.LParam);
```

Das Datenelement KBHandlerWnd

Das Datenelement *KBHandlerWnd* wurde von der Klasse *TApplication* entfernt. Die Eingabebehandlung ist über die virtuelle Elementfunktion *PreProcessMsg* von *TWindow* implementiert.

MAXPATH

In ObjectWindows 1.0 war MAXPATH in der Header-Datei filewnd.h definiert, in ObjectWindows 2.5 nicht mehr. MAXPATH ist in der Header-Datei dir.h definiert. Wenn Sie deshalb die Definition MAXPATH verwenden, sollten Sie jetzt die Standard-Header-Datei dir.h einbinden.

Stilkonventionen

ObjectWindows 2.5 verwendet etwas andere Stilkonventionen als ObjectWindows 1.0. Auch wenn Ihre Anwendung ohne diese stilistischen Änderungen korrekt compilierbar sein sollte, sollten Sie diese Änderungen dennoch vornehmen, um eine leichte Kompatibilität mit Ihrem zukünftigen Code unter ObjectWindows sicherzustellen.

Ändern von WinMain in OwlMain

Bei ObjectWindows 1.0 verwendeten Sie die Funktion *WinMain*, um eine Instanz der Klasse *TApplication* zu erzeugen, und riefen deren Elementfunktion *Run* auf. Bei ObjectWindows 2.5 erledigen Sie dies in der Funktion *OwlMain*. ObjectWindows 2.5 bietet ein vorgegebenes *WinMain*, das die Behandlung von Fehlern und Exceptions durchführt. Die Standardfunktion *WinMain* ruft die Funktion *OwlMain* auf. Wenn Sie irgendwelche Initialisierungen haben, die Sie in *WinMain* vorgenommen hatten,

sollten Sie diese nach *OwlMain* verlegen und Ihre *WinMain*-Funktion entfernen.

OwlMain unterscheidet sich von *WinMain* in ihrem Kopf. Wo *WinMain* eine Zahl von Windows-spezifischen Argumenten erwartet, nimmt *OwlMain* ein **int** und ein **char **** auf und gibt ein **int** zurück – genauso wie die Funktion *main* in einem traditionellen C- oder C++ Programm.

Sie müssen noch Ihre eigene Anwendungsklasse von *TApplication* ableiten, um *InitMainWindow* und *InitInstance* zu überschreiben. Der Konstruktor von *TApplication* erfordert von Ihnen nicht mehr die Angabe der Instanzen-Handles, Kommandozeilen und das Show-Flag des Hauptfensters; die versteckte Funktion *WinMain* sorgt selbst für die Bereitstellung dieser Werte (Sie können den Namen optional angeben).

Hier ist ein Beispiel für die Verwendung der Funktion *OwlMain*:

```
class TMyApp: public TApplication {
public:
  TMyApp(char far *name): TApplication(name) {}
  void InitMainWindow();
  };

void TMyApp::InitMainWindow() {
  .
  .
  .
  }

int OwlMain(int argc, char* argv[]) {
  return TMyApp("Wow!").Run();
  }
```

Datentypen und Namen

Funktionen in ObjectWindows 2.5 verwenden Typennamen von Windows wie LPSTR, PWORD und HANDLE nur dann, wenn eine direkte Verbindung zwischen diesem Element und irgendetwas in der Windows-API gibt. Ein Beispiel dazu ist die Verbindung zwischen der Funktion zur Behandlung von Ereignissen und der Windows-Botschaft, die sie behandelt. ObjectWindows 2.5 vermeidet auch soweit möglich die Verwendung von Typen im Windows-Stil wie PTWindowsObject und RTMessage und benutzt stattdessen C++ Typnamen wie **char far ***, **unsigned short *** und **const void ***. Dies trägt dazu bei, die Konventionen von ObjectWindows von der Windows-API zu abstrahieren, und erleichtert zukünftige Portierungsprobleme auf andere Umgebungen.

Ferner wurden Funktionsparameter in ObjectWindows 1.0 üblicherweise mit A*Something* benannt; d.h.: der Name hatte als Präfix ein großes A, der erste Buchstabe des Namens war groß geschrieben und der Rest des Namens klein. ObjectWindows 2.5 verwendet einen klein geschriebenen Namen ohne das große A als Präfix.

Der Konstruktor von *TWindow* sah in ObjectWindows 1.0 zum Beispiel wie folgt aus:

```
TWindow(PTWindowsObject AParent, LPSTR ATitle, ⋮ );
```

Der Konstruktor von *TFrameWindow* in ObjectWindows 2.5 (das Äquivalent des *TWindow*-Konstruktors in ObjectWindows 1.0) sieht wie folgt aus:

```
TFrameWindow(TWindow *parent, const char *title, ⋮ );
```

Beachten Sie, daß die Typen PTWindowsObject und LPSTR zu *TWindow* * und **const char *** verändert wurden, während die Parameternamen *AParent* und *ATitle* zu *parent* und *title* geändert wurden.

OWLCVT führt diese Konvertierungen für Sie durch. Wenn Sie allerdings nicht sorgsam umgehen, kann dies zu Problemen führen, da die Konvertierung nur die erste Instanz einer in einer Zeile deklarierten Variablen betrifft. Nehmen wie zum Beispiel an, Sie haben die folgende Deklaration:

```
PTEdit ptEdit1, ptEdit2, ptEdit3, ptEdit4;
```

Nach der Konvertierung würde dies wie folgt aussehen:

```
TEdit _FAR * ptEdit1, ptEdit2, ptEdit3, ptEdit4;
```

Damit sind *ptEdit2*, *ptEdit3*, und *ptEdit4* in Wirklichkeit *TEdit*-Instanzen und nicht Zeiger auf *TEdit*-Dialogelemente. Sie können diese Problem beheben, indem Sie die Zeil so ändern, daß jede Instanz des Zeigertyps in einer separaten Zeile vorkommt:

```
PTEdit ptEdit1;
PTEdit ptEdit2;
PTEdit ptEdit3;
PTEdit ptEdit4;
```

Alternativ dazu können Sie die Zeile nach dem Ablauf von OWLCVT korrigieren, indem Sie den Operator * bei jedem Variablennamen hinzufügen.

```
TEdit _FAR * ptEdit1, * ptEdit2, * ptEdit3, * ptEdit4;
```

Ersetzen von MakeWindow durch Create

ObjectWindows 2.5 verwendet die Funktion *TWindow::Create*, um Fenster zu erzeugen, und nicht die Funktion *TModule::MakeWindow* wie in ObjectWindows 1.0. Auch wenn die Funktion *Create* in ObjectWindows 1.0 existierte, bot *MakeWindow* einen sichereren Weg, ein Fenster zu erzeugen. Diese Funktion führte nämlich einen bestimmten Umfang an Fehlerüberprüfung durch bevor *Create* aufgerufen wurde. Diese Überprüfungen fehlten beim alleinigen Aufruf von *Create*. In ObjectWindows 2.5 werden allerdings die C++ Exceptions ausgenutzt, um solche Fehler aufzufangen, ohne den in *MakeWindow* enthaltenen expliziten Code zur Fehlerbe-

handlung zu verwenden. Es wird von Ihnen nicht gefordert, *Create* an Stelle von *MakeWindow* zu verwenden; *MakeWindow* existiert noch und kann wie zuvor ohne Änderungen im Code benutzt werden. Es wird allerdings als veraltet betrachtet und wird wahrscheinlich in zukünftigen Versionen der ObjectWindows-Klassenbibliothek entfernt werden.

Ersetzen von ExecDialog durch Execute

ObjectWindows 2.5 verwendet die Funktion *TDialog::Execute* Funktion an Stelle der in ObjectWindows 1.0 allgemein verwendeten Funktion *TModule::ExecDialog*. Dabei spielen dieselben Gründe wie bei der Verwendung von *Create* an Stelle von *MakeWindow* im vorherigen Abschnitt eine Rolle. Wie *TModule::MakeWindow* existiert auch *TModule::ExecDialog* noch und kann wie zuvor verwendet werden. Es wird allerdings als veraltet betrachtet und wird wahrscheinlich in zukünftigen Versionen der ObjectWindows-Klassenbibliothek entfernt werden. Ein Beispiel:

```
(new TDialog(MainWindow, "DIALOG_1"))->Execute();
```

Ermitteln der Anwendungs- und Modulinstanz

Die Anwendungs- und Modulinstanz wurde in die Bibliotheksverwaltung von ObjectWindows 2.5 gekapselt. Dies ermöglicht eine einfache Manipulation der von Borland und von Benutzern definierten DLLs. Um diese Änderung zu erleichtern, sollten Sie Aufrufe der Funktion *GetApplication()->hInstance* durch einen Aufruf von *GetLibInstance* ersetzen. Nehmen wie zum Beispiel an, Sie haben den folgenden Code:

```
Cursor = LoadCursor(GetApplication()->hInstance, "ThisCursor");
```

Sie können dies wie folgt konvertieren:

```
Cursor = LoadCursor(GetLibInstance(IDL_APPLICATION), "ThisCursor");
```

Definieren von WIN30, WIN31 und STRICT

Sie müssen WIN30, WIN31 oder STRICT nicht definieren, so lange Sie owldefs.h einbinden (oder eine Datei, die owldefs.h einbindet, wie owl.h) bevor Sie windows.h einbinden. Die Header-Datei owldefs.h definiert STRICT und bindet windows.h für Sie ein. Wenn Sie aber windows.h vor owldefs.h einbinden, müssen Sie STRICT definieren. Ebenso können Sie nur Windows 3.1 oder nachfolgende Versionen mit ObjectWindows 2.5 anvisieren, indem Sie die Verwendung des Makros WIN30 eliminieren.

Fehlerquellen

Dieser Abschnitt listet eine Reihe allgemeiner Probleme, die während der Konvertierung Ihres Codes von ObjectWindows 1.0 nach ObjectWindows 2.5 auftreten können.

OWLCVT-Fehler

Dieser Abschnitt beschreibt einige allgemeine Warnungen und Fehlermeldungen, die beim Ablauf von OWLCVT in Verbindung mit Ihrem ObjectWindows 1.0-Code auftreten können. Einige dieser Meldungen werden auf dem Bildschirm angezeigt, während OWLCVT Ihren Code verarbeitet. Andere Meldungen werden wiederum als Kommentare in Ihrer konvertierten Dateien plaziert.

- **Nicht erkannter DDVT-Wert**
  OWLCVT hat für einen DDVT-Wert keine spezielle Übersetzung. In diesem Falle fügt es einen generischen Wert ein, den Sie suchen und manuell ersetzen können.
- **Sicherheitskopie kann nicht erzeugt werden**
  OWLCVT erzeugt Sicherheitskopien aller Quelltext- und Header-Dateien, die es verändert, und legt sie im Verzeichnis OWLBACK ab. Wenn Sie diese Warnung erhalten, konnte OWLCVT aus irgendeinem Grund keine Sicherheitskopien erzeugen.
- **Redeklaration von *var***
  Dies ist zu einem Compiler-Fehler äquivalent, der besagt, daß Sie das Datenelement *var* redeklariert haben.

Compiler-Warnungen

Hier sind einige allgemeine Warnungen, die beim Durchlauf Ihres konvertierten ObjectWindows 1.0-Codes durch den Compiler von Borland C++ 4.5 auftreten können:

- **Paint versteckt Funktion**
- **ShutDownWindow versteckt Funktion**
- **CloseWindow versteckt Funktion**
- **DestroyWindow versteckt Funktion**
- **IdleAction versteckt Funktion**

  Für jede dieser Funktionen können Sie eine Warnung erhalten, die der folgenden ähnlich ist:

```
Paint(HDC, PAINTSTRUCT &) versteckt virtual Paint(void *, void *)
```

Dies kann ignoriert werden: die (**void ***, **void ***)-Funktionen waren ein Teil des Borland-Mechanismus für die Unterstützung der Kompatibilität zwischen Windows 3.0 und 3.1. Diese Funktionen wurden niemals verwendet.

Compiler-Fehler

Hier sind einige allgemeine Fehler, die beim Durchlauf Ihres konvertierten ObjectWindows 1.0-Codes durch den Compiler von Borland C++ 4.5 auftreten können:

- **Der Typ LPSTR oder der Typ X muß eine Struktur oder Klassenname :: GetClassName sein**

  OWLCVT konvertiert Aufrufe in das Windows-API, indem es dem Aufruf den Operator **::** voranstellt. Wenn Sie den Namen einer API-Funktion in einem anderen Kontext als zum Aufruf einer Windows- API-Funktion verwenden (zum Beispiel zum Überschreiben der Elementfunktion *GetClassName* von *TWindow*), kann OWLCVT auch dort den Operator **::** hinzufügen (es gibt allerdings ein paar Fälle, die OWLCVT erkennt und damit ignoriert werden). Dies kann dazu führen, daß der Compiler einen Fehler generiert. Sie können diesen Fehler beheben, indem Sie den durch OWLCVT zugefügten Operator **::** wieder entfernen.

- **'TWindow *' kann nicht in 'T*Class* *' konvertiert werden**

  Dies wird durch die Tatsache hervorgerufen,daß *TWindow* in ObjectWindows 2.5 als virtuelle Basis verwendet wird. Sie können ein *TWindow* oder ein Zeiger auf *TWindow* nicht direkt in eine Klasse umwandeln, die virtuell von *TWindow* abgeleitet ist. Verwenden Sie das Makro DYNAMIC_CAST zur Behebung des Fehlers.

- **'Base *' kann nicht in 'Derived *' umewandelt werden**

  Verwenden Sie das Makro DYNAMIC_CAST, um den Zeiger auf Base in einen Zeiger auf Derived umzuwandeln. Dies ist im wesentlichen derselbe Fehler wie der vorangehende.

- **'int *' kann nicht in 'TScrollerBase *' umgewandelt werden**

  Sie müssen die Header-Datei scroller.h einbinden. In ObjectWindows 1.0 wurde dies durch owl.h erledigt. Die Verzeichnisse der Header-Dateien und das Layout haben sich für ObjectWindows 2.5 geändert.

Laufzeitfehler

Hier sind einige allgemeine Fehler, die beim Ablauf einer Anwendung auftreten können, die durch Compilieren eines ObjectWindows 1.0-Codes entstand:

- **Paint wird nicht aufgerufen**

  Die Deklaration der Funktion *Paint* hat sich geändert. Sie müssen Ihre Funktion *Paint* ändern, damit sie mit der Elementfunktion *Paint* von *TWindow* übereinstimmt.

- **BeforeDispatchHandler, AfterDispatchHandler werden nicht aufgerufen**

- **FirstThat oder ForEach arbeiten nicht**

  Es ist wichtig, typensicher zu bleiben, wenn Sie mehrfache Vererbung und virtuelle Basisklassen verwenden wie es auch in ObjectWindows 2.5 geschieht. Der Grund dafür liegt darin, daß bei mehrfacher und virtueller Vererbung die Adresse von Unterobjekten nicht immer dieselbe

ist wie die von Objekten, in denen sie liegen. Nehmen wie zum Beispiel an, wir haben in ObjectWindows 1.0 einen Zeiger auf ein *TDialog* und wir wollen einen Zeiger auf dessen Basisklasse *TWindowsObject* erhalten. Der folgende Code würde in ObjectWindows 1.0 arbeiten, obwohl er nicht typensicher ist, da wir unsere Konvertierung mit einem Zeiger auf **void** vorgenommen haben:

```
TDialog * dialog_pointer;
void * void_pointer;
WindowsObject * winObj_pointer;
..
void_pointer = (void *)dialog_pointer;
winObj_pointer = (TWindowsObject*)void_pointer;
```

In ObjectWindows 2.5 würde dies nicht funktionieren. Sie müßten die Konvertierung typensicher gestalten:

```
TWindow * window_pointer = (TWindow *)dialog_pointer;
```

Wenn der Compiler weiß, daß er einen Zeiger auf *TDialog* konvertieren soll, um auf eine virtuelle Basis zu zeigen, paßt er den Wert des Zeigers geeignet an. Diese Art unsicherer Typumwandlung kann in ObjectWindows 1.0-Code existieren, ohne den Code zu zerstören.

Hier ist ein Beispiel dafür. In diesem Beispiel legt *IsChild* über *FirstThat* fest, ob ein übergebenes **void *** aktuell ein untergeordnetes Fenster ist:

```
BOOL TMyWindow::IsChild(void * child) {
  if (FirstThat(Test, child))
     return TRUE;
  else return FALSE;
  }
```

Dabei ist die Funktion *Test* wie folgt:

```
BOOL Test(void * winChild, void * child) {
return winChild == child;
}
```

Angenommen, *IsChild* wurde mit einem Zeiger auf *TDialog*-Objekt aufgerufen. Dann würde dieser Code nicht korrekt compiliert werden. Nach der Änderung in eine Übergabe von *TWindow* * funktioniert es. Wenn Sie dies nach ObjectWindows 2.5 konvertieren, erwartet die Funktion *Test* ein *TWindow* * und nicht ein **void ***. Dies schlägt aus folgendem Grund fehl. Wenn *IsChild* mit einem Zeiger auf *TDialog* aufgerufen wurde, wurde er nach **void *** konvertiert. Die Testfunktion vergleicht dies mit *TWindow* * auf unsichere Weise. Aber die Funktion wird nicht arbeiten, da bei ihrem Aufruf ein *TDialog* * übergeben wurde. Auch wenn *TDialog* ein untergeordnetes Fenster war, stimmt dessen Zeigerwert mit keinem der *TWindow*-Zeiger in der Liste untergeordneter Fenster überein.

- **MDI-Anwendung hat keine Menüeinträge aktiviert**

  Stellen Sie sicher, daß Sie die Include-Datei MDI.RH von ObjectWindows 2.5 verwenden. Diese Datei enthält die Konstanten für Standardeinträge im MDI-Menü wie CM_CASCADECHILDREN. Verwenden Sie insbesondere nicht die Definitionen der Include-Datei owlrc.h aus ObjectWindows 1.0.

Index

!

-LangID, Option *556*
< Operator *76*
== Operator *75*

A

Ableiten neuer Klassen *27*
Ableiten von TApplication *41*
Abstrakte Funktionen *31*
Abstrakte Klassen *29*
ActivationResponse Funktion *662*
ActiveChild Funktion *654*
Add (TArray) Funktion *60*
AddItem (TVbxControl) Funktion *442*
AddLine (Funktion) *113*
AddStream (TDocument) Funktion *297*
AddString *318*
AddString (TListBox) Funktion *270*
After
  TPlacement Aufzählungstyp *88*
AfterDispatchHandler Funktionen *662*
Aktionsarten *157*
Aktivierung von Befehlseinträgen *209*
Allgemeine Dialogfenster *33, 67*
  Ausführen *274*
  Datei öffnen *70*
  Datei sichern *71*
  Farbauswahl *83*
  Konstruieren *273*
  Modalität *274*
  Verwenden *273ff*
Allgemeine Dialogfenster für Datei öffnen *70*
  TData-Elemente *277*
  Verwenden *277*
Allgemeine Dialogfenster für Datei sichern *71*
  TData-Elemente *278*
  Verwenden *278*
Allgemeine Dialogfenster für Drucker
  TData-Elemente *281*
  Verwenden *281*
Allgemeine Dialogfenster für Farbauswahl *83*
Allgemeine Dialogfenster für Farbe *275*
  TData-Elemente *275*
Allgemeine Dialogfenster für Schrift
  TData-Elemente *276*
  Verwenden *275*
Allgemeine Dialogfenster für Suchen und Ersetzen
  Verwenden *279*
Ändern des Menüeintragstexts *211*
Ändern eines Stiftes *54*
Ändern existierender Templates *290*
Ändern von CanClose *179*
Ändern von Header-Dateien *645*
ANSI C++ Standard
  Änderungen am *629*
Antwortdateien
  Für Konvertierung verwenden *629*
Antworten auf Benachrichtigungsbotschaften *640*
Antworten auf Botschaften mit untergeordneter Fenster-ID *639*
Antworten auf generelle Botschaften *640*
Antworten auf Kommandobotschaften *638*
Antworttabellen *193, 195, 197, 199, 201*
  Definieren *43, 65*
  Deklarieren *43*
  Hinzufügen *43, 117*
  Hinzufügen von Einträgen *65*
Antworttabellen:Definieren *117*
Anwendung *595, 597, 599, 601, 603, 605, 607, 609, 611, 613, 615, 617, 619, 621, 623, 625*
  Automatisierung *595, 597, 599, 601, 603, 605, 607, 609, 611, 613, 615, 617, 619, 621, 623, 625*
  Dok/View *559*
  Initialisierung *169*
  Laufzeitverwaltung *169*
  Neue erzeugen *454*
  Registrieren *560*
  Registrierung *128*
  Schließen *169*
  Umwandeln in Container *455*
  Umwandeln in Server *455*
Anwendung,Automatisierung *596ff*
Anwendungen

Makros zum Erzeugen von *547, 560*
OLE-, aktualisieren *556*
Schreiben *454*
Anwendungsinstanz
Ermitteln *667*
Anwendungsklasse *566*
Anwendungsklassen *36*
Anwendungsobjekt *41, 501*
Anwendungsinstanz ermitteln *112*
Dok/View unterstützen *119*
Anwendungsobjekte *169, 171, 173, 175, 177, 179, 549, 560*
Erzeugen *588*
Makros zum Erzeugen von *547, 560*
Portieren auf Server *559*
Portieren in Server *588*
portieren zu Servern *529*
Registrieren *592*
Registrieren als OLE-SERVER *583*
Anwendungsregistriertabelle *598*
Anzeige
Neuaufbau *556*
Anzeigen
Hinzufügen einer Anzeige zu einem View *304*
AppDictionary *508*
AppExpert *485, 486*
AppReg *598*
AppReg (Struktur) *506, 507, 517, 599, 600*
Arbeiten mit Klassenhierarchien *27*
Arbeiten mit Views *300*
argc Parameter *169*
argv Parameter *169*
Array-Iteratoren
Erzeugen *60, 75*
Verwenden *60, 72*
Arrays
Ableiten von *74*
Erzeugen *59*
Tlines *75*
Verwenden *60*
AssignMenu (TDecoratedFrame) Funktion *89*
AssignMenu (TFrameWindow) Funktion *65, 648*
AssignMenu Elementfunktion
TFrameWindow *255*
Assoziieren von Ereignisbezeichnern mit Ereignisbehandlungsfunktionen *64*
Assoziieren von Oberflächenobjekten mit Dialogelementen *270*
Attr (TFrameWindow) Datenelement *648*
Attr Datenelement
TFrameWindow *255*
Attr.Style *184*
Ausdrucksobjekte
Anzeigen weiterer Seiten *377*
Drucken *376*
Fensterinhalt *375*
Konstruieren *373*
Seitenaufteilung *376*
Übersicht *371, 376*
Ausführen allgemeiner Dialogfenster *274*
Ausführen eines Dialogfensters *263*
Ausführen eines Eingabedialogfensters *55*
Ausschneiden und Einfügen *562*
Auswählen von Grafikobjekten für einen Gerätekontext *54*
AUTODATA (Makro) *601*
AUTOFUNC (Makro) *601*
AUTOITERATOR (Makro) *623*
Automatische Subklasseneinordnung
Bibliothek für 3D-Dialogelemente von Microsoft *180*
Automatisierung *455ff*
Automatisierungs-Server *552*
Controller *455, 457*
Object *457*
Typbibliothek *455*
Automatisierungs-Kontroller *596*
Automatisierungs-Server *596, 597, 598*
Funktionsoptimierung *611ff*
Automatisierungsdefinition
Datentypbezeichner *607*
Automatisierungsdeklaration *600*
Automatisierungsobjekt
OLE *612*
Automatisierungsverbindungen *484*
Automatition
Server *455*
Automatsierungsdefinition *600*

B

Basisklasse
Initialisieren *43*
Bearbeiten *460*
Direktes *460*
Objekt *460*
Bearbeitung von WM_PAINT in TWindow *116*
Befehlaktivierung
Text des Menüeintrags verändern *211*

Befehlsaktivierung
Aktivierung von Befehlseinträgen *209*
Aktivierungsbefehle *207*
Befehlsaktivierungsbotschaften *203*
Befehlsaktivierungsschnittstelle *206*
Checked enum *208, 213*
Deaktivierung von Befehlseinträgen *209*
Deaktivierungsbefehle *207*
Empfängertest *208*
EV_COMMAND_ENABLE Makro *204*
Indeterminate enum *208, 213*
Konstruktion von Befehlsaktivierungsobjekten *207*
Prüfbefehlseinträge *208*
Prüfung ob Befehl abgearbeitet *208*
Setzen von Befehlseintragstext *207*
TButtonGadgetEnabler Klasse *206*
TCommandEnabler Klasse *204, 206*
TCommandEnabler Schnittstelle *206*
TMenuItemEnabler Klasse *206*
Umschalten von Befehlseinträgen *213*
Unchecked enum *208, 213*
WM_COMMAND_ENABLE Ereignis *203*
Before
TPlacement Aufzählungstyp *88*
BeforeDispatchHandler Funktionen *662*
BEGIN_REGISTRATION (Makro) *507*
BeginDocument
TPrintout Elementfunktion *377*
BeginDrucken
TPrintout Elementfunktion *377*
Behandeln des Kommandos »Weitersuchen« *281*
Behandeln von VBX-Dialogelementbotschaften *436*
Behandlung von Botschaften und Ereignissen *635*
Behandlung von Windows-Ereignissen *42*
Beispielprogramme
AutoCalc *488*
AutoGen *488*
CallCalc *488*
CppOcf *488*
DllRun *488*
GuidGen *488*
Localize *488*
MdiOle *488*
OwlOcf *488*
Register *488*
RegTest *488*
SdiOle *488*
Step14-Step17 *488*
Tic Tac Toe *488*
Benachrichtigungsbotschaften
Antworten auf *640*
Benachrichtigungsbotschaten *199*
Bezeichner *549, 550*
IDCANCEL *55*
IDOK *55*
Bibliothek für 3D-Dialogelemente von Microsoft *180*
Bibliothek für Dialogelemente im Borland-Stil (BWCC) *179, 180*
Bilden des Rahmens und des Client *652*
Bildlaufleisten *326*
BOCOLE (Bibliothek) *599*
BOCOLE.DLL *467, 468, 469, 470*
Borderstyle (TGadget) Aufzählungstyp *85*
Botschaften *471*
auf der ID untergeordneter Fenster basierend *639, 640*
Behandlen *635*
Benachrichtigung *640*
Dialogelement-Benachrichtigungs-Codes *639*
general *640*
generelle *663*
Kommando *638, 664*
spezialisierte Antworten *270*
Verarbeiten von Botschaften zum Suchen und Ersetzen *280*
Verwenden von DefWndProc für registrierte *658*
WM_DROPFILES *121*
WM_LBUTTONDOWN *44, 50*
WM_LBUTTONUP *50*
WM_MOUSEMOVE *50*
WM_OWLVIEW *120, 123, 124*
WM_PAINT *62, 72, 116*
WM_RBUTTONDOWN *44*
Botschaften mit untergeordneter Fenster-ID *640*
Antworten auf *639*
Botschaftsverarbeitung von Anwendungen *177*
Gesonderte Botschaftsverarbeitung *177*
Botschaftsverarbeitungsfunktionen
TApplication *655*
Botschaftsverarbeitungsfunktionen von TApplication *655*
Bottom
TLocation Aufzählungstyp *89*
BWCCEnabled (TApplication) Funktion *180*

C

CanClose (TApplication) Funktion *178, 179*
CanClose (TWindow) Funktion *46, 267*
  Rückgabewerte *69*
CapsLock
  TModeIndicator Aufzählungstyp *85*
Checkliste zur Konvertierung *633, 634*
ChildList Oberflächenobjekt-Datenelement *188*
Client
  Benachrichtigung *557*
Client-Fenster *564*
  erzeugen *564, 565*
Client/Server
  Client-Fenster *565*
  Client-Kompatibilität *587*
  Dictionary *559*
  Einrichten *569*
  Portieren auf OLE-Server *559*
  Portieren in OLE-Server *568*
  Starten *593*
client/server Anwendungen *553*
Client/Server-Anwendungen *547, 549, 550*
  einrichten *501*
  Erzeugen von *547*
  portieren zu OLE-Servern *529*
Clienten-Benachrichtigung *556*
Close (Funktion) *110*
Close (TDocument) Funktion *297*
CloseWindow (TDialog) Funktion *264, 267*
CloseWindow Funktion *658*
CM_PENSIZE Makro *77*
CmAbout Funktion *73*
CmAbout-Funktion *121*
CmCancel (TDialog) Funktion *264, 267*
CmdShow Parameter *176*
CmFileNew *102*
CmFileNew (TDocManager) Funktion *120*
CmFileNew Funktion *62, 66*
CmFileOpen *103*
CmFileOpen Funktion *70*
CmFileSave Funktion *69*
CmFileSaveAs Funktion *70*
CmOk (TDialog) Funktion *264, 267*
CmPenColor Funktion *82*
CmPenColor-Funktion *117*
CmPenSize Funktion *77*
CmPenSize-Funktion *117*
Command
  TType Aufzählungstyp *87*
Commit *140*
Commit (TDokument) Funktion *110, 112*
Compiler-Fehler *668*
Compiler-Warnungen *668*
Compilieren und Linken *558*
Compilierung *558*
ComponentFactory (Funktion) *533*
Container *125, 148, 167, 455, 457*
  berwechseln von Object-basierten zur BIDS-Bibliothek *649*
  erzeugen *512*
  OLE *499ff*
  OLE- *124*
  Server und *552*
ContainerGroup enum *257*
Containerklasse TArray *58*
  Ableiten von *74*
  Konstruieren *59*
Containerklasse TArrayIterator *58*
  Konstruieren *60*
Count (Funktion) *625*
Create (TDialog) Funktion *263, 265*
Create Funktion *666*
Create Oberflächenobjekt-Funktion *183*
CreateAnyDoc (TDocManager) Funktion *292*
CreateAnyView (TDocManager) Funktion *292*
CreateDoc (TDocTemplate) Funktion *122*
CreateOcView
  Methode *565*
CreateOleObject (Funktion) *516*
CreateOleObjekt
  Methode *566*
Ctl3dEnabled (TApplication) Funktion *180*
Current (TArrayIterator) Funktion *61*

D

Datei-Menü
  Dokumentverwaltung *287*
Datentypbezeichner
  Automatisierungsdefinition *607*
DC-Funktionen *385*
DC-Klassen *35*
DDLs *149*
DDVTs
  Konvertieren *635*
Deaktivierung von Befehlseinträgen *209*
DECLARE_AUTOCLASS (Makro) *600*
DECLARE_STREAMABLE Makro *650*

DefChildProc
Konvertieren *657*
DefCommandProc
Konvertieren *657*
DEFINE_APP_DICTIONARY
Makro *560*
DEFINE_APP_DICTIONARY (Makro) *503*, *515*
DEFINE_APP_DICTIONARY Makro *547*
DEFINE_AUTOCLASS (Makro) *605*
DEFINE_AUTOENUM (Makro) *610*
DEFINE_DOC_TEMPLATE_CLASS Makro *288*
DEFINE_DOC_TEMPLATE_CLASS-Makro *118*
DEFINE_RESPONSE_TABLE Makros *43*
DEFINE_RESPONSE_TABLE-Makros *117*
Definieren einer Antworttabelle *43*, *65*, *117*
Definitionsmakro *606*
DefNotificationProc
Konvertieren *657*
DefWndProc
Konvertieren *657*
Deklarationsmakro *601*
Deklarieren einer Antworttabelle *43*
Dekorationen *34*
Dekorierte Fenster *33*
Dekorierte Rahmenfenster
Einfügen von Objekten in *89*
erzeugen *119*
Hinzufügen eines Menü-Deskriptors *119*
Konstruieren *84*
Dekoriertes Rahmenfenster
Menü-Mitverfolgung *85*
Destroy Funktion *658*
DestroyWindow Windows-API-Funktion *265*
DetachStream (TDocument) Funktion *297*
Dialogelemente *313*, *315*, *317*, *319*, *321*, *323*, *325*, *327*, *329*, *331*, *333*, *335*, *337*, *339*, *341*
Änderen der Attribute *316*
Anzeigen *317*
Assoziieren mit Oberflächenobjekten *270*
Bildlaufleisten *326*
Bildlaufleisten konstruieren *327*
Bildlaufleisten kontrollieren *327*
Bildlaufleisten, abfragen *328*
Bildlaufleisten, ändern *328*
Bildlaufleisten, antworten auf *328*
Bildlaufleisten, Positionszeiger mitverfolgen *329*
Bildlaufleistenbereich *327*
Dialogfenster und *318*
Editierfelder *330*
Editierfelder abfragen *332*
Editierfelder ändern *332*
Editierfelder konstruieren *330*
Editierfelder, Menü Bearbeiten *331*
Editierfelder, Zwischenablage *331*
Einrichten *271*
Entfernen *317*
Fenster und Transferpuffer *339*
Gleitregler *330*
Gruppenfelder *326*
Gruppenfelder antworten auf *326*
Gruppenfelder konstruieren *326*
Gruppieren *326*
Initialisieren *315*, *317*
Klassenname von *313*
Kombinationsfenster *333*
Kombinationsfenster abfragen *335*
Kombinationsfenster ändern *335*
Kombinationsfenster auswählen *334*
Kombinationsfenster konstruieren *335*
Kombinationsfenster, verschiedene Formen *333*
Kommunikation mit *269*
Kommunizieren mit *317*
Konstruieren *315*
Konstruktorparameter *316*
Listenfelder *318*
Listenfelder abfragen *320*
Listenfelder ändern *319*
Listenfelder konstruieren *319*
Listenfelder, antworten auf *320*
Manipulieren *318*
Markierungsfelder *324*
Markierungsfelder abfragen *325*
Markierungsfelder ändern *325*
Markierungsfelder konstruieren *324*
Meßanzeigen *330*
Schalter *323*
Schalter, antworten auf *323*
Schalter, konstruieren *323*
Schaltfelder *324*
Schaltfelder konstruieren *324*
statische *321*
statische ändern *322*
statische, abfragen *323*
statische, konstruieren *321*
TransferData *340*
Transfermechanismus *336*
Transferpuffer *336*
Transferpuffer definieren *337*
Transferpuffer, Dialogfenster *339*
Transferpuffer, Kombinationsfenster *338*

Transferpuffer, Listenfelder *338*
Werte einstellen und lesen *336, 337, 338, 339, 340, 341, 342*
Zeiger auf *315*
Dialogelementklassen *34*
Dialogelementleiste
Erzeugen *86*
Größe *87*
Schalter-Gadgets *86*
Separator-Gadgets *88*
TControlBar *86*
Dialogelementleisten
Einfügen von Gadgets in *88*
Dialogelementobjekte *270, 313ff*
Dialogelementobjekte von Visual Basic *431ff*
Dialogelementwerte *336ff*
Dialogfenster
allgemein *70*
allgemeine *67, 71, 83*
Allgemeine ausführen *274*
allgemeine für Datei öffnen *277*
allgemeine für Datei sichern *278*
allgemeine für Drucker *281*
allgemeine für Farbe *275*
allgemeine für Schrift verwenden *275*
allgemeine für Suchen und Ersetzen *279*
Allgemeine konstruieren *273*
Allgemeine verwenden *273ff*
allgemeine, Modalität *274*
als ein Client-Fenster *262*
Ausführen *263*
Automatische Subklasseneinordnung als 3D-Dialogelemente *180*
Fehlerbehandlung *266*
Info- *154*
Konstruieren *262*
Manipulieren von Dialogelementen in *269*
modal *263*
nicht-modal *263*
Schließen *267*
TXInvalidChildWindow Exceptions *267*
TXInvalidWindow Exceptions *267*
Übergeordnetes Fenster *262*
Verwalten *266*
Verwenden *272*
Verwenden als das Hauptfenster *655*
Verwenden als Hauptfenster *268*
Verwenden Eingabe *272*
von ObjectWindows gekapselt *272*
Dialogfensterklassen *33*
Dialogfensterobjekte *261ff*
Verwenden *261, 262, 263, 264, 265, 266, 267*
DisableAutoCreate (TWindow) Funktion *265*
DispatchAMessage Funktion *663*
DLL *547*
mehrere Prozesse *559*
DLL Server *456*
DLL-Server *547*
Compilieren und Linken *590*
Debuggen *592*
EXE-Server im Vergleich *587*
Laden und Starten *593*
Prozesse überwachen *560*
Registrieren von Anwendungen *592*
Server *587*
Zeitgeber *587*
DllEntryPoint *444*
DLLs *443, 445, 447, 449, 451, 547*
Dictionary *559*
Eintritts- und Austrittsfunktionen *444*
Funktionen exportieren *446*
Funktionsaufruf *443*
importieren *446*
Klassen und *447*
LibMain *444*
Makro _OWLDLL *449*
schreiben *443, 444, 445, 446*
Statische Daten *444*
WEP *445*
DLLs,DllEntryPoint *444*
DocReg *598*
Variable *566*
Dok/View
Modell *559*
Portieren auf Server *559*
Dok/View-Eigenschaftsattribute *310*
Dok/View-Modell *106, 499*
in einer Anwendung unterstützen *119*
Dok/View-Objekte *283ff*
Dokument *90*
Dokument-Templateklassen entwickeln *118*
Dokument-Templates *566, 571*
Ändern existierender Templates *290*
Dokument-Templateklassen entwickeln *118*
ein Dokument aus Templates erzeugen *122*
Entwurf von Dokument-Template-Klassen *288*
Erzeugen von Instanzen einer Template-Klasse *289*
Flags zur Templateklasse *119*
passende *122*

Template-Klassen-Flags *289*
Templateklassen-Instanzen erzeugen *118*
Dokumenten-Darstellungsklassen *36*
Dokumentklasse
Datenzugriffsfunktionen *296*
Erzeugen *295*
Stream-Zugriff *296*
Dokumentmanager
erzeugen *119*
passende Dokument-Templates *122*
suchen *112*
Dokumentobjekte
Änderungen in den Views registrieren *112*
Dok/View-Eigenschaftsattribute *310*
Eigenschaften *309*
Ermitteln von Dok/View-Eigenschaften *311*
Erweitern der Dokumentfunktionalität *299*
Setzen von Dok/View-Eigenschaften *311*
Zugreifen auf Eigenschaftsinformationen *311*
Dokumentregistrierstrukturen *598*
Dokumentverwaltung
Arbeiten mit der Dokumentverwaltung *300*
Arbeiten mit Views *300*
Datei-Menü *287*
Komplexer Datenzugriff *297*
Kostruieren *292*
MDI-Modus *292*
SDI-Modus *292*
Verwenden *291, 292, 293, 294*
Down
TState Aufzählungstyp *87*
DragAcceptFiles (TWindow) Funktion *120*
DragFinish (TDropInfo) Funktion *122*
DragQueryFile (TDropInfo) Funktion *122*
DragQueryFileCount (TDropInfo) Funktion *121*
DragQueryFileNameLen (TDropInfo) Funktion *121*
Draw Funktion *80, 81*
Draw-Funktion *116*
DrawMenuBar Elementfunktion
TWindow *251*
Drucken
mit ObjektWindows *371*
Drucken in einen Gerätekontext *48*
Drucker
Auswählen *373*
Konfigurieren *378*
mehrere *372*
Druckergeräte
Auswählen *378*
Druckerobjekte
Angeben von Druckergeräten *378*
Auswählen von Druckergeräten *378*
Konstruieren *372, 373*
mehrere *372*
Übersicht *371*
Verwenden des Standarddruckers *372*
Druckklassen *36*
Durchführen grafischer Operationen *48, 51*
Dynamische Linkbibliotheken *443, 445, 447, 449, 451*

E

EditGroup enum *257*
Editierfelder *330*
Verbinden mit Prüfungsobjekten *427*
Eigenschaften
automatisierbare *600*
Definition externer *604*
Dok/View-Eigenschaftsattribute *310*
Einbetten *457*
Einfügen von Gadgets in eine Dialogelementleiste *88*
Einfügen von Objekten in einen dekorierten Rahmen *89*
Einführung
Ändern der Liniendicke *53*
Ändern von Stiften *79*
Behandlung von Windows-Ereignissen *42*
Dateien *40, 42, 48, 50, 53, 58, 67, 74, 79, 84, 105*
Dateien in der *40*
Grundlegende Anwendung *40*
Hinzufügen mehrerer Linien *74*
Hinzufügen von Dekorationen *84*
Schreiben in ein Fenster *48*
Übergang zum Dok/View-Modell *105ff*
Verwenden allgemeiner Dialogfenster *67*
Zeichnen des Fensters und Hinzufügen von Menüs *58*
Zeichnen in das Fenster *50*
Eingabe
Filterung *428*
Eingabedialogfenster
Verwenden *272*
Eingabeprüfungen *423, 425, 427, 429*
Einrichten von Dialogelementen *271*
Embossed
Borderstyle (TGadget) Aufzählungstyp *85*

EnableAutoCreate (TWindow) Funktion *265*
EnableBWCC (TApplication) Funktion *180*
EnableCtl3d (TApplication) Funktion *180*
EnableCtl3dAutosubclass (TApplication) Funktion *180*
END_AUTOCLASS (Makro) *605*
EndDocument
  TPrintout Elementfunktion *377*
EndDrucken
  TPrintout Elementfunktion *377*
Entfernen des Oberflächenelementes *186*
Entfernen eines Gerätekontextes *384*
Entfernen von Fenstern *190*
Entfernen von Oberflächenobjekten *186*
Entwurf von Dokument-Template-Klassen *288*
Ereignisantworttabellen *635*
Ereignisbehandlung *117, 161f, 193ff, 305f, 307f*
  TDocManager *293*
Ereignisbehandlungsfunktionen *44*
  Assoziieren mit Ereignisbezeichnern *64*
  Extrahieren von Botschaften *45*
  Hinzufügen *66*
  Menükommandos *64*
Ereignisbezeichner
  Assoziieren mit Ereignisbehandlungsfunktionen *64*
  Hinzufügen *64*
  Menükommandos *64*
Ereignisse
  Behandeln *635*
  Behandlung im Anwendungsobjekt *306*
  Behandlung in einem View *305, 306, 307, 308*
  Hinzufügen eigener View-Ereignisse *308*
  in einem View behandeln *117*
Ermitteln der Anwendungs- und Modulinstanz *667*
Ermitteln des Pinselursprungs *386*
Ermitteln von Information über einen Gerätekontext *385*
Error
  TValidator Elementfunktion *429*
Erzeugen
  Objekte *549, 561*
  OLE-Objekte *552*
  Server-Anwendungen *558*
Erzeugen einer Dialogelementleiste *86*
Erzeugen einer Dokumentklasse *295*
Erzeugen einer Fensterklasse *42*
Erzeugen einer Statuszeile *85*
Erzeugen einer View-Klasse *302*
Erzeugen von Instanzen einer Template-Klasse *289*
Erzeugen von Menüdarstellungen *256*
Erzeugen von Oberflächenobjekten *183, 184, 185*
Erzeugen von Schalter-Gadgets *86*
Erzeugen von untergeordneten Oberflächenelementen *189*
EV_COMMAND Makro *64, 65*
EV_LBN_DBLCLK Makro *639*
EV_OWLNOTIFY Makro *300*
EV_VBXEVENTNAME Makro *432, 436*
EV_VN_COMMIT-Makro *117*
EV_VN_REVERT-Makro *117*
EV_WM_DROPFILES-Makro *120*
EV_WM_LBUTTONDOWN Makro *44, 50*
EV_WM_LBUTTONUP Makro *50*
EV_WM_MOUSEMOVE Makro *50*
EV_WM_RBUTTONDOWN Makro *44, 50*
Eval (Funktion) *605, 609*
EvCloseView-Funktion *124*
EvDropFiles-Funktion *121*
EvLButtonDown Funktion *61*
EvLButtonDown-Funktion *117*
EvLButtonUp-Funktion *117*
EvMouseMove Funktion *62*
EvMouseMove-Funktion *117*
EvNewView-Funktion *123*
EvPaint (TWindow) Funktion *62*
EvRButtonDown-Funktion *117*
EvVbxDispatch (TVbxEventHandler) Funktion *435*
Exception Handling *217*
Exception-Klasse
  TXBase *218*
  TXOwl *220*
Exception-Objekte
  klonen *219*
Exceptions
  TXInvalidChildWindow *267*
  TXInvalidWindow *267*
Exclusive
  TType Aufzählungstyp *87*
ExecDialog Funktion *667*
Execute (TChooseColorDialog) Funktion *83*
Execute (TDialog) Funktion *55, 73, 121, 263, 274*
Execute (TFileOpenDialog) Funktion *70*
Execute (TFileSaveDialog) Funktion *71*
Execute Funktion *667*
Execute Oberflächenobjekt-Funktion *183*

EXPOSE (Makro) *607*
EXPOSE_APPLICATION (Makro) *607*
EXPOSE_DELEGATE (Makro) *612*
EXPOSE_ITERATOR (Makro) *624*
EXPOSE_METHOD (Makro) *618*
EXPOSE_PROPRW (Makro) *618*
EXPOSE_xxxx (Makro) *605*
ExtendSelection
  TModeIndicator Aufzählungstyp *85*
Extractor-Operatoren (*650*

F

Factory *534, 598*
Factory Callback *508*
Fehler
  Compiler *668*
  Laufzeit *669*
  OWLCVT *668*
Fehlerbehandlung
  Dialogfenster *266*
Fehlerquellen *667, 668, 669, 670, 671, 672*
Fenster *33*
  Dekorierte Rahmen *84*
  dekorierte Rahmenfenster *119*
  dekorierter Rahmen *85, 89*
  Drucken *375*
  Durchführen grafischer Operationen *48, 51*
  Entfernen *190*
  Leeren *49*
  Schließen *46*
  Schreiben in *48*
  Ungültig machen *49*
  verbergen *150*
  Zeichnen *58*
  Zeichnen in *50, 54*
Fenster-Handle *154*
  Gültigkeit *183*
Fenstereigenschaften *185*
Fensterklasse
  Erzeugen *42*
Fensterklassen *32*
Fensterkonstruktoren
  Konvertieren *643*
Fensterobjekt
  Benennen *53*
Fensterobjekte *227, 229, 231, 233, 235, 237, 239, 241, 243, 245, 247*
  Assoziieren mit Oberflächenelementen *228*
  basierend auf TWindow *228*
  Konstruieren *228*
  Konstruieren von Rahmenfenstern *240*
  Verwenden *227, 228, 229, 230, 231*
FileGroup enum *257*
Finden eines Anwendungsobjektes *171*
FirstThat Funktionen *659*
FirstThat Oberflächenobjekt-Funktion *188, 191*
Flags
  AnyRegOptions *155*
  WB_MDICHILD *653*
  WS_VISIBLE *266*
Flags zur Template-Klasse *119*
Flush (TArray) Funktion *60, 66*
ForEach Funktionen *659*
ForEach Oberflächenobjekt-Funktion *188, 191*
Funktion
  LibMain *169*
  OwlMain *169, 172*
  WinMain *169, 172*
Funktionen
  ActivationResponse *662*
  AddLine *113*
  AfterDispatchHandler *662*
  BeforeDispatchHandler *662*
  Close *110*
  CloseWindow *658*
  CmAbout *73, 121*
  CmFileNew *62, 66, 102*
  CmFileOpen *70, 103*
  CmFileSave *69*
  CmFileSaveAs *70*
  CmPenColor *82, 117*
  CmPenSize *77, 117*
  Commit *140*
  Create *666*
  Destroy *658*
  DispatchAMessage *663*
  Draw *80, 81, 116*
  Ereignisbehandlung *44, 45*
  EvCloseView *124*
  EvDropFiles *121*
  EvLButtonDown *45, 48, 54, 61, 117*
  EvLButtonUp *50, 117*
  EvMouseMove *50, 62, 117*
  EvNewView *123*
  EvRButtonDown *45, 54, 117*
  ExecDialog *667*
  Execute *667*
  FirstThat *659*

ForEach *659*
GetApplication *164*
GetCommandTarget *154*
GetDocument *153*
GetFileData *99, 104*
GetItemsInContainer *113*
GetLine *113, 116*
GetModule *656*
GetPenSize *77*
InitChild *104*
InitMainWindow *119*
IsEmbedded *153*
IsOpen *138*
IsOptionSet *150*
MakeWindow *666*
MessageBox *46, 69*
Open *108, 140*
OpenFile *71, 77*
OwlMain *41, 119*
Paint *62, 78, 82, 116, 144, 167, 658*
QueryColor *80*
QueryPen *74*
QueryPenSize *80*
SaveFile *72, 77*
SetOcApp *134*
SetPen *81*
SetPenSize *56*
ShutDownWindow *658*
TApplication-Botschaftsverarbeitung *655*
VBXInit *431*
VBXTerm *431*
VnCommit *117*
VnRevert *117*

G

Gadgets *343ff*
Gadgetfenster *357f*
Gadgetklassen *351ff*
Schalter-Gadgets *86*
Separator-Gadgets *88*
GDI
Gerätekontexte *382*
GDI-Hilfsklassen *35*
GDI-Objekte *35*
Gekapselte API-Aufrufe *46*
Generelle Botschaften *663*
Antworten auf *640*
Gerätekontexte *382*
Auswählen von Grafikobjekten für *54*
Drucken in *48*
Entfernen *52*
Ermitteln von Information über *385*
Konstruieren *48, 51*
Pinselursprung ermitteln *386*
Pinselursprung setzen *386*
Speichern *385*
von ObjectWindows gekapselt *35*
Wiederherstellen *385*
Zurücksetzen *385*
Gesonderte Botschaftsverarbeitung *177*
GetActiveChild Funktion *654*
GetAppDescriptor (Funktion) *613*
GetApplication (TApplication) Funktion *112*
GetCommandTarget *154*
GetDCOrg (TDC) Funktion *385*
GetDeviceCaps (TDC) Funktion *385*
GetDialogInfo
TPrintout Elementfunktion *376*
GetDocManager (TApplication) Funktion *112*
GetDocManager (TDocument) Funktion *292, 300*
GetDocument *153*
GetEventIndex (TVbxControl) Funktion *438*
GetEventName (TVbxControl) Funktion *438*
GetFileData *99, 104*
GetItemsInContainer (Funktion) *113*
GetItemsInContainer (TArray) Funktion *60, 72*
GetLine (Funktion) *113*
GetLine-Funktion *116*
GetMainWindow (TApplication) Funktion *65, 89, 119, 121*
GetModule Funktion *656*
GetNumEvents (TVbxControl) Funktion *438*
GetNumProps (TVbxControl) Funktion *439*
GetPenSize Funktion *77*
GetProp (TVbxControl) Funktion *440*
GetPropIndex (TVbxControl) Funktion *440*
GetPropName (TVbxControl) Funktion *439*
GetViewMenu (TView) Funktion *123*
GetWindow (TView) Funktion *123*
Grafikklassen *35*
Grafikobjekte *379ff*
Gruppieren von Pop-Up Menüs *258*
GUIDGEN.EXE (Hilfsmittel) *504*

H

HasNextPage
  TPrintout Elementfunktion *373*
HasPage
  TPrintout Elementfunktion *377*
Hauptfenster
  Verwenden eines Dialogfensters als *268, 655*
Hauptfenster-Anzeigemodus *176*
  Ändern *177*
Header
  einbinden *611*
Header-Datei
  Rahmenfenster *92*
Header-Datei applicat.h *170*
Header-Datei owl\module.h *170*
Header-Dateien *148*
  MDI *91*
  ObjectWindows *646*
  OLE-Klassen *127*
Header-Dateiverzeichnisse *645*
HelpGroup enum *257*
Hilfselemente (Widgets) *34*
Hilfsfunktionen zum Zeichnen *386*
hInstance Parameter *169*
Hinzufügen der Definition einer Ereignisantworttabelle *637*
Hinzufügen der Deklaration einer Ereignisantworttabelle *637*
Hinzufügen der Ereignisbezeichner *64*
Hinzufügen eigener View-Ereignisse *308*
Hinzufügen einer Antworttabelle *43, 117*
Hinzufügen einer Anzeige zu einem View *304*
Hinzufügen einer Fensterklasse *42*
Hinzufügen eines Menüs zu einem View *303*
Hinzufügen eines Stiftes *53*
Hinzufügen von Antworttabelleneinträgen *65*
Hinzufügen von Einträgen für die Ereignisantworttabelle *638*
Hinzufügen von Ereignisbehandlungsfunktionen *66*
Hinzufügen von Funktionalität zu Dokumenten *295*
Hinzufügen von Funktionalität zu Views *303*
Hinzufügen von Menü-Ressourcen *65*
Hinzufügen von Menüs *58*
Hinzufügen von TEditFile-Client-Fenstern *661*
Hinzufügen von TEditSearch-Client-Fenstern *661*
Hook *595, 596, 603*
Horizontal
  TTileDirection Aufzählungstyp *86*
hPrevInstance Parameter *169, 174*
HWindow (TDialog) Datenelement *271*
HWindow Oberflächenobjekt-Datenelement *183*

I

ID-Strings *549, 550*
IDCANCEL *55*
IDE *558*
  Erzeugen von Server-Anwendungen *590*
Idle-Verarbeitung *177*
IdleAction (TApplication) Funktion *177*
IDOK *55*
ifstream Klasse *71*
IMPLEMENT_STREAMABLE Makro *650*
Implementieren virtueller TDocument-Funktionen *108*
Implementieren von Streaming *650*
Include-Dateien
  OLE-Server *558*
Include-Pfad
  Für Konvertierung verwenden *629, 631, 645*
Indeterminate
  TState Aufzählungstyp *87*
Init (TFrameWindow) Funktion *262*
Init (TWindow) Funktion *43*
InitApplication (TApplication) Funktion *41, 171, 174*
InitChild *104*
Initialisieren Basisklasse *43*
Initialisieren des Hauptfensters *42, 175*
Initialisieren des Stiftes *53*
Initialisieren eines 16-Bit-Anwendungsobjektes *174*
Initialisieren eines 32-Bit-Anwendungsobjektes *174*
Initialisieren eines Anwendungsobjektes *171, 174*
Initialisieren von Anwendungsinstanzen *175*
Initialisierung der ersten Instanz *174*
InitInstance (TApplication) Funktion *41, 120, 171, 175*
InitMainWindow *151*
InitMainWindow (TApplication) Funktion *42, 65, 84, 120, 171, 268*
InitMainWindow-Funktion *119*
Insert (TControlBar) Funktion *88*

Insert (TDecoratedFrame) Funktion *89*
insertable *157*
Inserter-Operatoren (<) *650*
Instantiierung von Klassen *28*
InStream (TDocument) Funktion *296*
Integrated Development Environment *479*
Invalidate (TWindow) Funktion *49, 66, 72, 275*
InvalidateObject (Funktion) *613*
InvalidateRect (TWindow) Funktion *49*
InvalidateRgn (TWindow) Funktion *49*
IsOK (TView) Funktion *123*
IsOpen *138*
IsOptionSet *150*
IsValid
  TValidator Methode *428*
IsValidInput
  TValidator Elementfunktion *428*
IsWindowVisible
  Oberflächenobjekt-Funktion *185*

K

Kapselung
  Anwendung *169, 171, 173, 175, 177, 179*
  DLL *169*
  hInstance *169*
  hPrevInstance *169*
  lpCmdLine *169*
  nCmdShow *169*
KBHandlerWnd Datenelement *664*
Klasse
  TGadgetWindowFont *86*
Klassen *471*
  ifstream *71*
  MDI *651*
  ofstream *72*
  string *112, 660*
  TApplication *41, 169, 291*
  TArray *58, 59, 74*
  TArrayIterator *58, 60, 72*
  TButtonGadget *86*
  TChooseColorDialog *83*
  TChooseColorDialog::TData *83*
  TClientDC *48*
  TColor *79*
  TControl *433*
  TControlBar *86*
  TDC *48, 383*
  TDecoratedFrame *84, 92, 119*
  TDecoratedMDIFrame *92, 127*
  TDialog *73, 121*
  TDocManager *106, 119, 291, 292, 293, 294*
  TDocument *138*
  TDrawDocument *107*
  TDrawMDIChild *96*
  TDrawView *114*
  TDropInfo *121*
  TEditFile *660*
  TEditSearch *660*
  TEditWindow *660*
  TFileDialog *662*
  TFileOpenDialog *70, 662*
  TFileSaveDialog *71*
  TFileWindow *660*
  TFindDialog *662*
  TFrameWindow *41*
  TGadgetWindow *88*
  TInputDialog *55*
  TInStream *109*
  TLine *74*
  TMDIChild *652*
  TMDIClient *92, 101, 652*
  TMDIFrame *92, 652*
  TMenuDescr *115, 119*
  TModule *169*
  TObjekt *106*
  TOleDocument *127, 138*
  TOleMDIFrame *127, 133*
  TOleView *127, 144*
  TOpenSaveDialog::TData *67*
  TOutStream *111*
  TPen *53, 54, 79*
  TReplaceDialog *662*
  TResId *73, 87*
  TSearchDialog *662*
  TSeparatorGadget *88*
  TStatusBar *85*
  TStorageDocument *138*
  TVbxControl *433*
  TVbxEventHandler *432, 435*
  TView *106, 114, 302*
  TWindow *114*
  TWindowView *114*
  Typen von Elementfunktionen *30*
  VBX-Dialogelement *432*
Klassenfabrik *129*
Klassenhierarchien *27*
Kollektionsklasse *620, 625*
Kombinationsfenster *333*

Kombinieren von Objektverhalten *28*
Kommandobotschaften *664*
  Antworten auf *638*
Kommandos
  Behandeln von »Weitersuchen« *281*
Kommandozeile *130, 131, 150*
Kommandozeilenoptionen *554*
Kommunikation mit Dialogelementen *269*
Konfigurationsdateien
  Für Konvertierung verwenden *629*
Konstruieren
  TApplication *170, 171, 172*
  TClientDC *48*
  TDC *384*
  TDocument *295*
  TFrameWindow *47, 240, 268*
  TVbxControl *433*
Konstruieren der Dokumentverwaltung *292*
Konstruieren eines Anwendungsobjektes *171, 172*
Konstruieren eines Gerätekontextes *48, 51, 384*
Konstruieren TButtonGadget *86*
Konstruieren TControlBar *86*
Konstruieren TStatusBar *85*
Konstruieren virtueller Basen *648*
Konstruieren von allgemeinen Dialogfenstern *273*
Konstruieren von Dialogfenstern *262*
Konstruieren von Fensterobjekten *228*
Konstruieren von Rahmenfensterobjekten *240*
Konstruieren von TView *302*
Konstruieren, TDocManager *292*
Konventionen
  Stil *664*
Konvertieren
  DefChildProc *657*
  DefCommandProc *657*
  DefNotificationProc *657*
  DefWndProc *657*
  Ersetzen von ActiveChild durch GetActiveChild *654*
  Ersetzen von ExecDialog durch Execute *667*
  Ersetzen von MakeWindow durch Create *666*
  MDI-Klassen *651*
  Object-basierte Container zur BIDS-Bibliothek *649*
  OWLCVT *631*
  Sicherheitskopien für Ihre alten Quelltextdateien *631*
  TWindowsObject/TWindow *642*
  von DDVTs nach Ereignisantworttabellen *635*
  Windows-API-Funktionen *644*
  WinMain nach OwlMain *664*
Konvertieren nach Borland C++ 4.5 *629*
Konvertieren von Fensterkonstruktoren *643*
Konvertieren von ObjectWindows Code *627ff*
Konvertierungs-Checkliste *633*
Konvertierungsprozeduren *635ff*
Konvertiren von Windows-Code nach STRICT *629*

L

large (Speichermodell) *500*
Laufzeitfehler *669*
Layoutbedingung *232*
Layoutfenster *232*
Layoutmetrik *232*
LCID (Attribut) *615*
Leeren des Fensters *49*
Left
  TLocation Aufzählungstyp *89*
LibMain *444*
LibMain Funktion *169*
LineTo (TWindow) Funktion *52*
Listen untergeordneter Fenster *188*
Locale-ID (Attribut) *615*
locale.h (header) *615*
Lokalisierung *456*
LookAt (Funktion) *605*
LookAtWindow (Funktion) *605*
Löschen von Oberflächenobjekten *187*
LPARAM Parameter *641*
LPARAM Variable *45*
lpCmdLine Parameter *169*

M

MainWindow Variable *654*
Make-Dateien
  Für Konvertierung Verwenden *629*
MAKELANGID (Makro) *618*
MakeWindow Funktion *666*
Makro
  BEGIN_REGISTRATION *583*
  BI_APP_DLL *588*
  DEFINE_APP_DICTIONARY *560*
  END_REGISTRATION *583*
  REGDATA *586*

Makro _OWLDLL *449*
Makroerweiterung
OLE-Server *553*
Makros
Benachrichtigungsbotschaften *199*
CM_PENSIZE *77*
DEFINE_DOC_TEMPLATE_CLASS *118, 288*
DEFINE_RESPONSE_TABLE *43, 117*
EV_COMMAND *64, 65*
EV_OWLNOTIFY *300*
EV_VBXEVENTNAME *432, 436*
EV_VN_COMMIT *117*
EV_VN_REVERT *117*
EV_WM_DROPFILES *120*
EV_WM_LBUTTONDOWN *44, 50*
EV_WM_LBUTTONUP *50*
EV_WM_MOUSEMOVE *50*
EV_WM_RBUTTONDOWN *44, 50*
MB_YESNOCANCEL *69*
NOTIFY_SIG *308*
RC_INVOKED *65*
Server registrieren *583*
VN_DEFINE *308*
WM_VBXFIREEVENT *436*
Zur Registrierung verwenden Server und Container dieselben Makros, aber Server müssen dabei mehr Informationen bereitste *549*
Manipulieren von Dialogelementen in Dialogfenstern *269*
Manipulieren von untergeordneten Fenstern *191*
MatchTemplate (TDocManager) Funktion *122*
Mauspalette *134, 161, 164*
MAXPATH *664*
MB_YESNOCANCEL Makro *69*
MDI *90*
Fensterklassen *95, 101*
Header-Dateien *91*
MDI-Anwendungen
ortieren auf Server *568*
Portieren auf Server *558*
MDI-Fenster *33*
MDI-Klassen *651*
MDI-Modus *292*
Mehrere Prozesse
DLLs und *547*
Menü *525*
einemRahmenfenster zuteilen *255*
Menü-Deskriptoren
erzeugen *115, 119*
Menü-Mitverfolgung *85*
Menü-Ressourcen *64, 65, 647*
Hinzufügen *65*
Menüdarstellungen
erzeugen *256*
Gruppierung von Pop-Up Menüs *258*
Menütrenner *258*
Menüdeskriptoren *256*
Gruppierung von Pop-Up Menüs *258*
Hinzufügen eines View *115*
Konstruktion von *257*
zu einem dekorierten Rahmenfenster hinzufügen *119*
Zusammensetzen von Menüdeskriptoren *259*
Menüelemente
TrackPopupMenu *252*
Menükommandos *64*
Assoziieren von Ereignisbezeichnern mit Ereignisbehandlungsfunktionen *64*
Ereignisbehandlungsfunktionen *64*
Ereignisbezeichner *64*
Ressourcen *64*
Menüobjekte *249, 251, 253, 255, 257, 259*
Abfrage von *251*
Erzeugen von Menüdarstellungen *256*
Gruppierung von Pop-Up Menüs *258*
Konstruktion *249*
Konstruktion von Menüdeskriptoren *257*
Menüdeskriptoren hinzufügen *256*
Modifizieren von *250*
Pop-Up Menüs *252, 253*
System *252*
Zusammensetzen von Menüdeskriptoren *259*
Menüs *37*
Hinzufügen eines Menüs zu einem View *303*
Hinzufügen zu einem Fenster *58, 65*
Zuweisen an ein Rahmenfenster *648*
Menütrenner *258*
MergeMenu (TFrameWindow) Funktion *123*
MessageBox (TWindow) Funktion *46*
Rückgabewerte *69*
MessageLoop (TApplication) Funktion *177*
Methode
automatisierbare *600*
CreateOcView *565*
CreateOleObjekt *566*
Methoden
Definition externer *604*
Modale Dialogfenster *263*
Modalität allgemeiner Dialogfenster *274*

Modulinstanz
Ermitteln *667*
Modulklassen *36*
Move (TVbxControl) Funktion *442*
Multiple Document Interface *90*

N

nCmdShow (TApplication) Datenelement *176*
nCmdShow Parameter *169*
Nicht erkannter DDVT-Wert *668*
Nicht-modale Dialogfenster *263*
Nichtvirtuelle Funktionen *31*
None
Borderstyle (TGadget) Aufzählungstyp *85*
NonExclusive
TType Aufzählungstyp *87*
NOTIFY_SIG Makro *308*
NotifyViews (TDocument) Funktion *300*
NotifyViews (TDokument) Funktion *112*
NumLock
TModeIndicator Aufzählungstyp *85*

O

Oberflächenelemente
Assoziieren mit Fensterobjekten *228*
Entfernen *186*
Sichtbarmachen *184*
Untergeordnete und übergeordnete *187, 188, 189, 190, 191*
Oberflächenobjekt
Löschen *187*
Oberflächenobjekte *181ff, 587*
Assoziieren mit Dialogelementen *270*
ChildList Elemente *188*
Create Elemente *183*
Eigenschaften *185*
Entfernen *186*
Erzeugen *183, 184, 185*
Execute Elemente *183*
FirstThat Elemente *188, 191*
ForEach Elemente *188, 191*
HWindow Elemente *183*
IsWindowVisible Elemente *185*
Parent Elemente *188*
SetupWindow Elemente *184*
Show Elemente *185*
ObjectCDomponents
TOcApp *471*
ObjectComponents *126, 149, 453ff, 595*
Anwendung erzeugen *479*
Anwendungen weitergeben *480*
AppExpert *485, 486*
Ausschnitt *474*
AutoGen *486*
Automatisierung *468*
Automatisierungsklassen *472*
Automatisierungsverbindungen *484*
Beispielprogramme *488*
Bibliotheken *480*
BOCOLE.DLL *481*
Botschaften *471, 473, 476, 477, 478, 479, 480*
Bücher *487, 488*
Deklarationstabellen *484*
DLL Server *468*
DllRun *486*
Drag-and-Drop *468*
Einbettende Klassen *471*
Ereignisse *162*
Exception Handling *477*
EXE Server *468*
Fehlercodes *478*
Fenster *476, 477, 478, 479, 480*
GuidGen *486*
Hilfsprogramme *486*
IBContainer *481*
IDataObject *481*
IDispatch *481, 484*
IMoniker *481*
ITypeLib *484*
IUnknown *481, 482*
Klassen *467, 468, 469, 470, 471, 484*
Lokalisierung *470*
MacroGen *486*
OC_APPBORDERSPACEREQ *474*
OC_APPBORDERSPACESET *474*
OC_APPCREATECOMP *474*
OC_APPDIALOGHELP *474*
OC_APPFRAMERECT *474*
OC_APPMENUS *474*
OC_APPPROCESSMSG *474*
OC_APPRESTOREUI *474*
OC_APPSHUTDOWN *474*
OC_APPSTATUSTEXT *474*
OC_VIEWATTACHWINDOW *474*
OC_VIEWBORDERSPACESET *474*
OC_VIEWCLIPDATA *474*
OC_VIEWCLOSE *474*

OC_VIEWDRAG *474*
OC_VIEWDROP *474*
OC_VIEWGETPALETTE *474*
OC_VIEWGETSCALE *474*
OC_VIEWGETSITEREC *474*
OC_VIEWINSMENUS *474*
OC_VIEWLOADPART *474*
OC_VIEWOPENDOC *474*
OC_VIEWPAINT *474*
OC_VIEWPARTINVALID *474*
OC_VIEWPARTSIZE *474*
OC_VIEWSAVEPART *474*
OC_VIEWSCROLL *474*
OC_VIEWSETSCALE *474*
OC_VIEWSETSITEREC *474*
OC_VIEWSHOWTOOLS *474*
OC_VIEWTITLE *474*
OCAPPINSMENUS *474*
OCVIEWBORDESPACEREQ *474*
Online-Hilfe *487*
Operationen der Zwischenablage *468*
Programmierhilfen *485, 486*
QueryInterface *481*
Register *486*
Registrierung *470*
Schnittstelle *481, 482*
TAutoBase *472*
TAutoFactory *476*
TAutoIterator *484*
TAutoProxy *472*
TOcApp *482*
TOcDocument *471*
TOcModule *471*
TOcPart *471, 482*
TOcRegistrar *471*
TOcRemView *471, 482*
TOcView *471, 482*
TOleAllocator *472*
TOleDocument *476*
TOleDocViewFactory *476*
TOleFactory *476*
TOleFrame *476*
TOleMDIFrame *476*
TOleView *476*
TOleWindow *476*
TRegistrar *472*
TServedObject *484*
TStorageDocument *476*
TTypeLibrary *484*
TUnknown *482*
TXAuto *477*
TXObjComp *477*
TxOle *477*
TXRegistry *477*
Typbibliotheken *470*
Verbinderobjekte *482*
Verknüpfen und Einbetten *468*
Verknüpfende Klassen *471*
Verknüpfungsobjekte *472, 482*
View *474*
WinRun *486*
WM_OCEVENT *473*
ObjectComponents-Framework *499*
ObjectGroup enum *257*
ObjectWindows *126, 149*
OLE-Container *513ff*
ObjectWindows Header-Dateien *646*
ObjectWindows-gekapselte Dialogfenster *272*
ObjectWindows-Ressourcen *646*
Objekt
Aktivieren *462*
Auswählen *462*
Bearbeiten *466*
Convert *463*
Copy *463*
Cut *463*
Deaktivieren *462*
Delete *463*
Edit *463*
Einfügen *459*
Größe verändern *462*
Konvertieren *463*
Koperen *463*
Linking *464*
Lokales Menü *463*
Löschen *463*
Objekt-Aktionsarten *463*
Offenes Bearbeiten *466*
Öffnen *463, 466*
Open *463*
Open editing *466*
Schneiden *463*
verändern *462*
Verknüpfen *464*
Verschieben *462*
Objekte
Server und *552*
Objektkollektionen *619*
Objekttypologie *32ff*
Öffnen einer Dokumentdatei *108*

Öffnen und Schließen einer Darstellung *108*
Öffnen und Speichern von Zeichnungen *71*
Öffnen vordefinierter DLLs *179, 180*
ofstream Klasse *72*
OLE *453ff*
  Aggregation *489*
  Aktionsart *497*
  Aktivieren *489*
  Anwendung *595ff*
  äußeres Objekt *489*
  Ausschnitt *490*
  Auswählen *490*
  automatisierte Anwendung *490*
  automatisiertes Objekt *490*
  Automatisierung *490, 595, 597, 599, 601, 603, 605, 607, 609, 611, 613, 615, 617, 619, 621, 623, 625*
  Automatisierungs-Controller *490*
  Automatisierungs-Server *490*
  Automatisierungsobjekt *612*
  BOCOLE Support Library *491*
  Client *491*
  COM Objekt *491*
  Container *491, 499ff*
  Direkte Bearbeitung *493*
  DLL Server *491*
  Dokument *492*
  Eingebettete Objekte *492*
  EXE-Server *492*
  Fernausschnitt *492*
  GUID *493*
  IDispatch-Schnittstelle *493*
  In-Process Server *493*
  IUnknown *489*
  IUnknown Interface *494*
  Lager *498*
  Lokalisierung *494*
  ObjectComponents Framework *494*
  ObjectWindows *513ff*
  ObjectWindows-Bibliothek *494*
  Offenes Bearbeiten *495*
  OLE-Schnittstelle *495*
  QueryInterface *489*
  Referenzzählung *495*
  Registrierung *495*
  Registrierungsdatenbank *496*
  Registrierungsobjekt *495*
  Registrierungstabelle *496*
  Schnittstelle *493*
  Server *496*
  Storage *498*
  System-Registrierungsdatenbank *496*
  Teil *497*
  TOcApp *495, 497*
  TOcDocument *490, 492*
  TOcPart *497*
  TOcRegistrar *495*
  TOcRemView *490, 492*
  TOcView *490, 497*
  TRegistrar *495*
  TRegList *496*
  Typenbibliotheken *496*
  Verbinderobjekt *497*
  Verknüpfende Objekte *498*
  Verknüpfungsquelle *497*
  Zusammengesetzte Datei *498*
  Zusammengesetztes Dokument *498*
OLE 2 *467, 468, 469, 470*
OLE-Anwendungen
  Aktualisieren *557*
  Benachrichtigungsbotschaften *549*
  Dictionary *559*
  Laden *557*
  Registrarobjekte *553*
OLE-Container *124, 499ff*
OLE-Header
  einbinden *512*
OLE-Klassen *548, 550*
  Bezeichner *549*
OLE-Objekte *549, 556*
  Speichern *557*
OLE-Server *147, 148, 545, 588*
  Dokumentliste *571*
  Flags *554, 555*
  Header-Dateien *567*
  Include-Dateien *558*
  konvertieren von Anwendungen *529*
  Makroerweiterung und *553*
  Prozesse überwachen *559*
  Registrieren von Anwendungen *549, 560, 562, 566, 583*
  Umwandeln von Anwendungen *559, 568*
  Umwandeln von Anwendungen *559*
olenls.h (header) *618*
OP_xxxx (Konstante) *609*
Open *140*
Open (Funktion) *108*
Open (TDocument) Funktion *297*
OpenFile Funktion *71, 77*
Operatoren *76*

< *76*
== *75*
OutStream (TDocument) Funktion *296*
Overtype
TModeIndicator Aufzählungstyp *85*
OWLCVT
Prozedur zum Konvertieren eines Programms *631*
OWLCVT Kommandozeilensyntax *631*
OWLCVT-Fehler *668*
OWLCVT-Konvertierungen *630*
OwlMain (Funktion) *518*
OwlMain Funktion *41, 169, 172*
OwlMain-Funktion *119*

P

Paint *144*
Drucken von Fenstern und *375*
PrintPage vs. *371*
Paint (TWindow) Funktion *62, 78, 82, 116*
Paint Funktion *62, 78, 82, 658*
Paint-Funktion *116*
Parent Oberflächenobjekt-Datenelement *188*
Pinselursprung
Ermitteln *386*
Setzen *386*
Plain
Borderstyle (TGadget) Aufzählungstyp *85*
Pop-Up Menü Objekte
TrackPopupMenu *252*
Portieren
STRICT definieren und *629*
Print
TPrinter Elementfunktion *375*
PrintPage
TPrintout Elementfunktion *371, 373, 376*
Prozeduren zur Konvertierung *635ff*
Prozesse
überwachen *547, 559*
Prüfungsobjekt
abstrakt *424*
TValidator *424*
Prüfungsobjekte
Bereich verwenden *424*
Bild *425*
Fehlerbehandlung *429*
Filter *424*
Konstruieren *427*
Nachschlagen *425*
String nachschlagen *425*
TFilterValidator *424*
Verbinden mit Editierfeldern *427*
Puffer
OLE-Server *553*

Q

QueryColor Funktion *80*
QueryPen Funktion *74*
QueryPenSize Funktion *80*
QueryViews (TDocument) Funktion *300*

R

Rahmenfenster *33, 93, 94*
Client-Fenster löschen *124*
Client-Fenster setzen *123*
Client-Menüs einarbeiten *123*
Fensterüberschrift setzen *124*
Header-Datei *92*
Rahmenmenü wiederherstellen *124*
Raised
Borderstyle (TGadget) Aufzählungstyp *85*
RC_INVOKED Makro *65*
Recessed
Borderstyle (TGadget) Aufzählungstyp *85*
RecordingMacro
TModeIndicator Aufzählungstyp *85*
Redeklaration von var *668*
Refresh (TVbxControl) Funktion *442*
REG.DAT
Speichern der Registrierungsdatenbank in *584*
REGDATA (Makro) *506, 517*
REGDOCFLAGS (Makro) *506*
REGFORMAT (Makro) *506*
Registrar *130, 150*
Registrar-Objekt *500, 507, 532*
Registrar-Objekte
OLE-Anwendungen *562, 571*
Registrarobjekt *599*
Registrarobjekte
OLE-Anwendungen *553, 554*
REGISTRATION_FORMAT_BUFFER (Makro) *506*
Registrierdatenbanken *571*
Registrieren als OLE-Server
Client/Server-Anwendungen *549*

Registrieren von Anwendungen
Server *588*
Registrierstrings *619*
Registriertabelle *128, 136, 156, 560, 597*
erzeugen *560*
Registriertabellen *619*
Registrierte Botschaften
Verwenden von DefWndProc für *658*
Registrierung *597*
Registrierungsdatenbanken *584*
Anlegen der Strukturen für *552*
Erzeugen der Tabellen für *553*
Erzeugung von Tabellen *549*
Schlüssel *550, 583, 586*
Server *586*
Speichern von Informationen *584*
Tabelle der Schlüssel *552*
Tabellen *561, 562*
Tabellen für *552*
Registrierungsmakro *508*
Registrierungsstruktur *508*
Registrierungstabellen *456, 516*
RegServer
Option *556*
Remote View
OLE-Server *564, 565*
RemoveItem (TVbxControl) Funktion *442*
ResetDC (TDC) Funktion *385*
Ressourcen
ObjectWindows *646*
Ressourcenskript *615*
Ressourcentabelle *596*
RestoreBrush (TDC) Funktion *386*
RestoreDC (TDC) Funktion *385*
RestoreFont (TDC) Funktion *386*
RestoreMenu (TFrameWindow) Funktion *124*
RestoreObject (TDC) Funktion *386*
RestorePalette (TDC) Funktion *386*
RestorePen (TDC) Funktion *386*
RestoreTextBrush (TDC) Funktion *386*
Revert (TDokument) Funktion *110, 112*
Right
TLocation Aufzählungstyp *89*
RTTI *608*
Rückwärtskompatibilität
Makros *552*
Run (Funktion) *500, 508*
Run (TApplication) Funktion *41*

S

SaveDC (TDC) Funktion *385*
SaveFile Funktion *72, 77*
Schalter-Gadgets
Bitmaps *87*
Ereignisbezeichner *87*
Erzeugen *86*
Gadget-Bezeichner *87*
Ressourcen-Bezeichner *87*
Schließen einer Anwendung *178*
Schließen eines Dialogfensters *267*
Schließen eines Dokumentes *299*
Schließen eines Fensters *46*
Schreiben in ein Fenster *48*
ScrollLock
TModeIndicator Aufzählungstyp *85*
SDI *90*
SDI-Anwendungen
Portieren auf Server *558, 567*
SDI-Modus *292*
Seitenaufteilung des Ausdruckes *376*
SelectObject (TDC) Funktion *54, 386*
SelectStockObject (TDC) Funktion *386*
SendDlgItemMessage (TWindow) Funktion *270*
Separator-Gadgets *88*
Server *125, 148, 457, 557, 558*
als unabhängige Anwendung benutzt *554*
Benachrichtigungsbotschaft *557*
Bibliotheken *558*
Container und *552*
Dokument-Templates *562*
egistrierung *566*
Einbettender *455*
erzeugen *611*
Linken *558*
Make-Dateien *558*
Objekte und *549, 552*
OLE *147, 148*
OLE- *556*
OLE-Server *558*
OWLOCFMK.GEN *558*
Portieren in *568*
Registrieren von Anwendungen *549, 552, 583*
Rückwärtskompatibilität bei Registrierung *552*
Templates *562*
Verknüpfen der *455*
SetCaption (TFrameWindow) Funktion *124*
SetClientWindow (TFrameWindow) Funktion
123, 124

SetDocManager (TApplication) Funktion *119, 291*
SetDocManager (TDocument) Funktion *300*
SetDocPath (TDokument) Funktion *108*
SetMainWindow (TApplication) Funktion *42, 47, 53, 89, 175, 177*
SetMenuDescr (TFrameWindow) Funktion *119*
SetMenuDescr Funktion *260*
SetOcApp *134*
SetPen Funktion *81*
SetPenSize Funktion *56*
SetPrinter
  TPrinter Elementfunktion *378*
SetPrintParams
  TPrintout Elementfunktion *376*
SetProp (TVbxControl) Funktion *441*
Setup
  TPrinter Elementfunktion *378*
SetupWindow *315*
  Aufruf *317*
SetupWindow (TDialog) Funktion *271*
SetupWindow Oberflächenobjekt-Funktion *184*
SetValidator
  TEdit Methode *427*
SetViewMenu (TView) Funktion *115*
Setzen des Pinselursprungs *386*
Setzen von Menüdeskriptoren eines Fensters *260*
Show
  Oberflächenobjekt-Funktion *185*
ShowWindow (TWindow) Funktion *263, 266*
ShutDownWindow (TDialog) Funktion *264, 267*
ShutDownWindow Funktion *658*
Sicherheitskopie kann nicht erzeugt werden *668*
Sicherheitskopien für Ihre alten Quelltextdateien *631*
Sichtbarmachen von Oberflächenelementen *184*
Speichermodelle *558, 590*
Speichern eines Gerätekontextes *385*
Speichern und Verwerfen von Änderungen *110*
Spezielle Dialogelemente *313ff*
Standarddialogelemente von Windows *34*
Standarddrucker
  Verwenden *372*
Standardplatzhalter-Funktionen *32*
StaticName (TView) Funktion *116*
Statuszeile
  Erzeugen *85*
  TStatusBar *85*
Stautszeile
  Ressourcen *85*
Stifte
  Ändern *54*
  Auswählen für einen Gerätekontext *54*
  Entfernen *57*
  Konstruieren *53*
  TPen Klasse *53*
Stilkonventionen *664*
Stream-Klassenbibliothek *650*
Streaming *650*
  Implementieren *650*
STRICT
  Definieren *667*
STRICT definieren *629*
String Klasse *660*
string-Klasse *112*
Strings
  erzeugen *112*
Struktur
  AppReg *597*
Strukturen
  TRect *301*
  VBXEVENT *436, 437*
Symbolnamen *614*

T

TAppDictionary (Klasse) *500*
TApplication
  Ableiten von *41*
  Anforderungen *170*
  Anwendungsinstanz ermitteln *112*
  BWCCEnabled Elemente *180*
  CanClose Elemente *178, 179*
  Ctl3dEnabled Elemente *180*
  Dok/View unterstützen *119*
  Elemente *41f, 47, 53, 65, 84, 89, 112, 119, 120f, 180*
  EnableCtl3d Elemente *180*
  EnableCtl3dAutosubKlasse Elemente *180*
  Ermitteln der Anwendungsinstanz *667*
  Finden des Objektes *171*
  GetApplication Elemente *171*
  Header-Datei *170*
  IdleAction Elemente *177*
  InitApplication Elemente *171, 174*
  Initialisieren *171, 174*
  InitInstance Elemente *171, 175*
  InitMainWindow Elemente *171, 268*

Konstruieren *171*
Konstruktoren *170, 171, 172*
MessageLoop Elemente *177*
nCmdShow Elemente *176*
Prozedur zum Schließen *179*
Run Elemente *172*
Schließen *178*
SetDocManager Elemente *291*
SetMainWindow Elemente *175, 177*
Übergeben von Kommandoparametern an *170*
Überschreiben *41*
TApplication (Klasse) *499, 501, 502, 515*
TApplication Klasse *41, 169, 291*
TApplication Konstruktoren
Übergeben von WinMain-Parametern *173*
TArray
Elemente *60, 66, 72*
TArray Klasse *58*
Ableiten von *74*
Konstruieren *59*
TArrayIterator
Elemente *61*
TArrayIterator Klasse *58*
Konstruieren *60, 72*
Tastendrücke
Überprüfen *428*
TAutoObject (Klasse) *603*
TAutoString (Klasse) *603*
TAutoVal (Klasse) *608*
TButton *324*
TButtonGadget Klasse *86*
Konstruieren *86*
TButtonGadgetEnabler Klasse *206*
TCalc (Klasse) *601*
TCalc (Makro) *608*
TCalcButtons (Klasse) *620, 625*
TCalendar (Klasse) *612*
TCheckBox *324*
TChooseColorDialog
Elemente *83*
TChooseColorDialog Klasse *83*
TChooseColorDialog::TData Klasse *83*
TClientDC Klasse *48*
TColor Klasse *79*
TComboBox *333*
TCommandEnabler Klasse *204*
Checked enum *208, 213*
Enable Funktion *207, 209*
GetHandled Funktion *208*
Handled Element *209*
HWNDReceiver Element *208*
Id Element *208*
Indeterminate enum *208, 213*
IsReceiver Funktion *208*
Konstruktion *207*
SetCheck Funktion *208, 213*
SetText Funktion *207, 211*
Unchecked enum *208, 213*
TControl *314*
TControl Klasse *433*
TControlBar
Elemente *88*
TControlBar Klasse *86*
Konstruieren *86*
TData Klasse
Allgemeine Dialogfenster *273*
Allgemeine Dialogfenster für Datei öffnen *277*
Allgemeine Dialogfenster für Datei sichern *278*
Allgemeine Dialogfenster für Drucker *281*
Allgemeine Dialogfenster für Farbe *275*
Allgemeine Dialogfenster für für Suchen und Ersetzen *279*
Allgemeine Dialogfenster für Schrift *276*
TData-Elemente für allgemeine Dialogfenster *273*
TDC
Destruktor *384*
Elemente *48, 54*
GetDCOrg Elemente *385*
GetDeviceCaps Elemente *385*
Konstruktoren *384*
ResetDC Elemente *385*
RestoreBrush Elemente *386*
RestoreDC Elemente *385*
RestoreFont Elemente *386*
RestoreObjects Elemente *386*
RestorePalette Elemente *386*
RestorePen Elemente *386*
RestoreTextBrush Elemente *386*
SaveDC Elemente *385*
SelectObject Elemente *386*
SelectStockObject Elemente *386*
TDC Klasse *48, 383*
TDecoratedFrame *92*
Elemente *89*
TDecoratedFrame (Klasse) *502*
TDecoratedFrame Klasse
Konstruieren *84*
Menü-Mitverfolgung *85*
TDecoratedFrame-Klasse

erzeugen *119*
TDecoratedMDIFrame *92, 127*
TDecoratedMDIFrame (Klasse) *502*
TDialog
CloseWindow Elemente *264, 267*
CmCancel Elemente *264, 267*
CmOk Elemente *264, 267*
Create Elemente *263, 265*
Elemente *55, 73, 121*
Execute Elemente *263, 274*
HWindow Elemente *271*
SetupWindow Elemente *271*
ShutDownWindow Elemente *264, 267*
UpdateData Elemente *280*
TDialog Klasse *73*
Konstruieren *73*
TDialog-Klasse *121*
erzeugen *121*
TDocManager
CreateAnyDoc Elemente *292*
CreateAnyView Elemente *292*
Elemente *120, 122*
Ereignisbehandlung *293*
TDocManager Klasse *291, 292, 293, 294*
TDocManager, Konstruktoren *292*
TDocManager-Klasse *106, 119*
TDocTemplate
Elemente *122*
TDocument *138*
AddStream Elemente *297*
Close Elemente *297*
DetachStream Elemente *297*
GetDocManager Elemente *292, 300*
Hilfsfunktionen für den Datenzugriff *298*
InStream Elemente *296*
Konstruktoren *295*
NotifyViews Elemente *300*
Open Elemente *297*
OutStream Elemente *296*
QueryViews Elemente *300*
SetDocManager Elemente *300*
TDocument (Klasse) *502*
TDokument
Elemente *108, 110, 112*
implementieren virtueller Funktionen *108*
Öffnen einer Dokumentdatei *108*
TDrawDocument aufbauen *107*
TDrawDocument entfernen *107*
TDrawDocument-Klasse *107*
TDrawMDIChild *96*
TDrawView-Klasse *114*
TDropInfo
Elemente *121, 122*
TDropInfo-Klasse *121*
TEditFile Klassen *279, 660*
TEditFile-Klassen
Hinzufügen von Client-Fenstern *661*
TEditSearch Klassen *279, 660*
TEditSearch-Klassen
Hinzufügen von Client-Fenstern *661*
TEditWindow Klassen *660*
Template *129*
Template-Klassen-Flags *289*
Templateklassen-Instanzen erzeugen *118*
TextOut (TDC) Funktion *48*
TFileDialog Klassen *662*
TFileDocument (Klasse) *502, 511*
TFileOpenDialog
Elemente *70*
TFileOpenDialog Klasse *70*
TFileOpenDialog Klassen *662*
TFileSaveDialog
Elemente *71*
TFileSaveDialog Klasse *71*
TFileWindow Klassen *660*
TFindDialog Klasse *281*
TFindDialog Klassen *662*
TFontListBox *314*
TFrameWindow
Elemente *47, 65, 119, 123, 124, 648*
Init Elemente *262*
Konstruktoren *240, 268*
Setzen des Fenstertitels *53*
TFrameWindow (Klasse) *502*
TFrameWindow Klasse *41, 643*
AssignMenu *255*
Attr Elemente *255*
SetMenuDescr Funktion *260*
TGadget
Elemente *85*
TGadgetWindow Klasse *88*
TGadgetWindowFont Klasse *86*
TGauge *330*
TGroup enum
ContainerGroup *257*
EditGroup *257*
FileGroup *257*
HelpGroup *257*
ObjectGroup *257*
WindowGroup *257*

THSlider *330*
TInputDialog Klasse *55*
  Ausführen *55*
  Ressourcen *55*
TInputValidator class *267*
TInputValidator Klasse *267*
TInStream-Klasse *109*
TLine Klasse *74*
Tlines Array *75*
TListBox *314, 316*
  AddString Elemente *270*
TListBoxData *338*
TLocaleString (Klasse) *618*
TLocation Aufzählungstyp *89*
TMDIChild Klasse *652*
TMDIClient *92, 101*
TMDIClient Klasse *652*
TMDIFrame *92*
TMDIFrame (Klasse) *502*
TMDIFrame Klasse *652*
TMenu Klasse *249, 251, 253, 255, 257, 259*
TMenuDescr (Klasse) *525*
TMenuDescr Klasse *249, 251, 253, 255, 256, 257, 259*
  Konstruktion *257*
  Zusammensetzen von Menüdeskriptoren *259*
TMenuDescr-Klasse
  erzeugen *115, 119*
TMenuItemEnabler Klasse *206*
TModeIndicator Aufzählungstyp *85*
TModule
  Ermitteln der Modulinstanz *667*
TModule Klasse *169*
TMyApp (Klasse) *501*
TObjektklasse *106*
TOcAutoFactory (Template) *600*
TOcDocument (Klasse) *530, 535*
TOcModule (Klasse) *499, 501, 515*
TOcRegistrar (Klasse) *507ff, 513, 533, 599*
TOcRegistrar* (Registrar) *508*
TOcView (Klasse) *530, 535*
TOleAllocator (Klasse) *530*
TOleDocument *127*
TOleDocument *138*
TOleDocument (Klasse) *502, 510*
TOleDocument (Klassse) *502*
TOleDocViewFactory *508*
TOleDocViewFactory (Template) *503*
TOleFactory (Klasse) *518*
TOleFrame (Klasse) *502, 507, 520, 522*
TOleMDIFrame *127, 133*
TOleMDIFrame (Klasse) *502, 520, 522*
TOleView *127, 144, 509*
TOleView (Klasse) *502*
TOleWindow (Klasse) *502, 521*
Top
  TLocation Aufzählungstyp *89*
TOpenSaveDialog::TData Klasse *67*
TOutStream-Klasse *111*
TPen
  Entfernen *57*
TPen Klasse *53, 79*
  Ändern *54*
  Konstruieren *53*
TPlacement Aufzählungstyp *88*
TPointer *599*
TPopUpMenu Klasse *249, 251, 252, 253, 255, 257, 259*
  TrackPopupMenu Element *252*
TPrinter Objekt
  Übersicht *371*
TPrinterDialog Objekt *378*
TPrintout Objekt *373*
  Übersicht *371*
TrackPopupMenu Funktion
  TPopupMenu *252*
TransferData *340*
Transferpuffer *336*
TRect Struktur *301*
TRegistrar (Klasse) *599*
TRegList (Struktur) *532*
TReplaceDialog Klasse *281*
TReplaceDialog Klassen *662*
TResId Klasse *73, 87*
TSearchDialog Klassen *662*
TSeparatorGadget Klasse *88*
TSlider *330*
TState Aufzählungstyp *87*
TStatic *321*
TStatusBar Klasse
  Konstruieren *85*
TStorageDocument *138*
TStorageDocument (Klasse) *511*
TSystemMenu Klasse *249ff*
TTileDirection Aufzählungstyp *86*
TType Aufzählungstyp *87*
TValidator *424*
TVbxControl
  AddItem Elemente *442*
  GetEventIndex Elemente *438*

GetEventName Elemente *438*
GetNumEvents Elemente *438*
GetNumProps Elemente *439*
GetProp Elemente *440*
GetPropIndex Elemente *440*
GetPropName Elemente *439*
Konstruktoren *433*
Move Elemente *442*
Refresh Elemente *442*
RemoveItem Elemente *442*
SetProp Elemente *441*
TVbxControl Klasse *433*
TVbxEventHandler
EvVbxDispatch Elemente *435*
Vermischen mit Oberflächenklassen *432*
TVbxEventHandler Klasse *432, 435*
TView
Elemente *115, 116, 123*
Ereignisbehandlung *117*
Hinzufügen einer Anzeige zu einem View *304*
Hinzufügen eines Menüs zu einem View *303*
Hinzufügen von Funktionalität *303*
Hinzufügen von Zeigern auf Oberflächenobjekte *304*
Kostruieren *302*
Schließen eines View *305*
Vermischen von TView mit Oberflächenobjekten *304*
Virtuelle Funktionen *303*
TView (Klasse) *502*
TView, Ereignisbehandlung *305, 306, 307, 308*
TView-Klasse *106*
Ableiten von *114*
TVSlider *330*
TWindow *314*
als das generische Oberflächenobjekt *182*
CanClose Elemente *267*
DisableAutoCreate Elemente *265*
Elemente *43, 46, 49, 52, 62, 66, 68, 69, 72, 78, 82, 116, 120*
EnableAutoCreate Elemente *265*
Invalidate Elemente *275*
SendDlgItemMessage Elemente *270*
ShowWindow Elemente *263, 266*
Verarbeiten von WM_PAINT *62, 82*
TWindow (Klasse) *502*
TWindow Klasse *643*
DrawMenuBar Element *251*
TWindow-Klasse
Ableiten von *114*
TWindowsObject/TWindow
Konvertieren *642*
TWindowView-Klasse *114*
Typbibliothek *625*

U

Überschreiben der Funktion CanClose *46*
Überschreiben der Funktion CanClose$ICanClose (TWindow) Funktion *68*
Überwechseln von Object-basierten Containern zur BIDS-Bibliothek *649*
Umschalten von Befehlseinträgen *213*
Umwandelbarkeit *650*
Umwandeln virtueller Basen zu abgeleiteten Typen *648*
Ungültig machen eines Fensters *49*
Untergeordnete und übergeordnete Oberflächenelemente *187, 188, 189, 190, 191*
Unterstützung von Dok/View in einer Anwendung *119*
Up
TState Aufzählungstyp *87*
UpdateData (TDialog) Funktion *280*

V

Valid
TValidator Methode *428*
Variable
DocReg *566*
MainWindow *654*
Variablen *547, 559*
VARIANT (Union) *603*
VBX-Dialogelemente
Abfragen der Dialogelementeigenschaften *440*
Behandeln von Botschaften *436*
Dialogelementmethoden *442*
Eigenschaften *439*
Ereignisantworttabelle *436*
Ereignisbehandlung *432*
Ermitteln von Eigenschaftsinformationen *439*
Ermitteln von Ereignisinformationen *438*
Implizites und explizites Konstruieren *434*
Interpretieren eines Dialogelementereignisses *437*
Verwenden *431*
Zugreifen *438*

VBX-Dialogelementklassen *432*
VBXEVENT Struktur *436, 437*
VBXInit Funktion *431*
VBXTerm Funktion *431*
Verarbeiten von Botschaften zum Suchen und Ersetzen *280*
Verarbeiten von WM_PAINT in TWindow *62, 82*
Verbunddatei *511*
Verbunddokumente *510*
Vererbung
  von OLE-Klassen *548*
Vererbung von Datenelementen *30*
Vererbung von Elementen *29*
Verhaltensänderung beim Schließen *178*
Verknüpfen *457*
Verknüpfen und Einbetten *457*
Vermischen von TView mit Oberflächenobjekten *304*
Vertical
  TTileDirection Aufzählungstyp *86*
Verwalten von Dialogfenstern *266*
Verwenden allgemeiner Dialogfenster *67*
Verwenden allgemeiner Dialogfenster für Datei öffnen *277*
Verwenden allgemeiner Dialogfenster für Datei sichern *278*
Verwenden allgemeiner Dialogfenster für Drucker *281*
Verwenden allgemeiner Dialogfenster für Farbe *275*
Verwenden allgemeiner Dialogfenster für Schrift *275*
Verwenden allgemeiner Dialogfenster für Suchen und Ersetzen *279*
Verwenden der Array-Klassen *61*
Verwenden der Bibliothek für 3D-Dialogelemente von Microsoft *180*
Verwenden der Bibliothek für Dialogelemente im Borland-Stil (BWCC) *180*
Verwenden der Dokumentverwaltung *291, 292, 293, 294*
Verwenden eines Fensters als ein Client *47*
Verwenden von allgemeinen Dialogfenstern *273, 274, 275, 276, 277, 278, 279, 280, 281, 282*
Verwenden von Dialogfenstern *272*
Verwenden von Dialogfensterobjekten *261, 262, 263, 264, 265, 266, 267*
Verwenden von Eingabedialogfenstern *272*
Verwenden von Fensterobjekten *227, 228, 229, 230, 231*
Verwenden von TApplication *170*
Verwenden von VBX-Dialogelemente *431*
Verwenden von WinMain und OwlMain *172*
Verwenden von WinMain-Parametern *173*
Verzeichnisse
  DLLs und *547*
  OLE-Anwendungen *547*
  OLE-Server *547*
View *152*
View schließen *124*
View-Klassen
  Erzeugen *302*
Views
  an ein Dokument anpassen *114*
  Änderungen im Dokument registrieren *112*
  Arbeiten mit *300*
  Dok/View-Eigenschaftsattribute *310*
  Eigenschaften *309*
  Ereignisbehandlung *117, 305, 306, 307, 308*
  Ermitteln von Dok/View-Eigenschaften *311*
  Hinzufügen eigener View-Ereignisse *308*
  Hinzufügen eines Menü-Deskriptors *115*
  Hinzufügen von Zeigern auf Oberflächenobjekte *304*
  Menü ausgeben *123*
  schließen *124*
  Setzen von Dok/View-Eigenschaften *311*
  Vermischen von TView mit Oberflächenobjekten *304*
  zugehöriges Fenster suchen *123*
  Zugreifen auf Eigenschaftsinformationen *311*
Virtuelle Basen
  Konstruieren *648*
  Umwandeln zu abgeleiteten Typen *648*
Virtuelle Funktionen *31*
Virtuelle Funktionen von TView *303*
VN_DEFINE Makro *308*
VnCommit-Funktion *117*
VnRevert-Funktion *117*
Vordefinierte Dok/View-Ereignisbehandlungen *307*

W

Warnungen
  Compiler *668*
WB_MDICHILD Flag *653*

WEP *444*
Werkzeugleisten *529*
Widgets (Hilfselemente) *34*
Wiederherstellen eines Gerätekontextes *385*
WIN30
  Definieren *667*
WIN31
  Definieren *667*
WindowGroup enum *257*
Windows Anwendungen
  Ereignisse *549*
Windows-Anwendungen
  32-Bit-EXE-Programme *587*
Windows-API-Funktionen *46, 644*
WinMain
  Verwenden von Parametern beim Konstruieren von Anwendungsobjekten *173*
WinMain Funktion *169, 172*
WM_CLOSE (Botschaft) *534*
WM_COMMAND_ENABLE Ereignis *203*
WM_DROPFILES (Botschaft) *121*
WM_LBUTTONDOWN Botschaft *44, 50*
WM_LBUTTONUP Botschaft *50*
WM_MOUSEMOVE Botschaft *50*
WM_OCEVENT (Botschaft) *522, 530*
WM_OWLVIEW-Botschaft *120, 123, 124*
WM_PAINT Botschaft *62, 72*
WM_PAINT-Botschaft *116*
WM_RBUTTONDOWN Botschaft *44*
WM_VBXFIREEVENT Makro *436*
WPARAM Parameter *641*
WPARAM Variable *45*
WS_THICKFRAME (Konstante) *534*
WS_VISIBLE
  Eigenschaft *184*
WS_VISIBLE Flag *266*

Z

Zeichnen eines Fensters *58*
Zeichnen in ein Fenster *50, 54*
Ziehen und Ablegen
  aktivieren *120*
  Informationen über eine abgelegte Datei ausgeben *121*
  Windows-Speicher freigeben *122*
Zugreifen auf Dokument- und View-Eigenschaften *311*
Zugreifen auf VBX-Dialogelemente *438*
Zugriff auf die Daten eines Dokumentobjekts *113*
Zugriffsoperator (::) *645*
Zurücksetzen eines Gerätekontextes *385*
Zusammensetzen von Menüdeskriptoren *259*
Zwischenablage *37, 158*

Borland

Copyright © 1994, Borland International, Inc. Alle Rechte Vorbehalten. Alle Produktnamen von Borland sind eingetragene Warenzeichen der Borland International Inc.
Deutsche Ausgabe © 1994 Borland GmbH, Monzastraße 4c, D-63225 Langen, Telefon 06103/979-0, Fax 06103/979-290
Zweigniederlassung Schweiz, Solothurnstraße 31, CH-3303 Jegenstorf
Hauptsitz: 100 Borland Way, Scotts Valley, CA 95066-3249, +1 (408) 431-1000
Niederlassungen in: Australien, Belgien, Brasilien, Chile, Deutschland, Dänemark, Frankreich, Großbritannien, Hongkong, Italien, Japan, Kanada, Korea, Lateinamerika, Malaysia, Niederlande, Neuseeland, Schweden, Singapur, Spanien und Taiwan.
• Part # BCP1245GExxxxx